

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

如何用函数式编程技术改进Java程序

Java函数式编程

Functional Programming in Java

[法] Pierre-Yves Saumont 著

高清华 译



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

内容简介

本书由法国著名程序员编写，是一本关于Java函数式编程的入门书籍。本书从函数式编程的基本概念出发，介绍了函数式编程的语法、语义、类型系统、编译原理、虚拟机、性能优化、应用案例等方面的知识。本书适合Java程序员、软件工程师、计算机科学专业的学生阅读。

本书是作者多年从事Java函数式编程工作的经验总结，内容翔实，重点突出。本书可作为Java函数式编程的教材，也可作为Java函数式编程的参考书。

本书共分10章，第1章介绍函数式编程的基本概念，第2章介绍函数式编程的语法，第3章介绍函数式编程的语义，第4章介绍函数式编程的类型系统，第5章介绍函数式编程的编译原理，第6章介绍函数式编程的虚拟机，第7章介绍函数式编程的性能优化，第8章介绍函数式编程的应用案例，第9章介绍函数式编程的进阶知识，第10章介绍函数式编程的总结。

本书可作为Java函数式编程的教材，也可作为Java函数式编程的参考书。

（CIP）数据

本书是作者多年从事Java函数式编程工作的经验总结，内容翔实，重点突出。本书可作为Java函数式编程的教材，也可作为Java函数式编程的参考书。

Java函数式编程

Functional Programming in Java

[法] Pierre-Yves Saumont 著

高清华 译

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

《Java函数式编程》并不是一本关于Java的书，而是一本关于函数式编程的书。作者由浅入深地介绍了函数式编程的思维方式，并引导读者通过易于掌握的例子、练习和图表来学习和巩固函数式编程的基本原则和最佳实践。读者甚至可以在阅读的同时编写出自己的函数式类库！

本书非常适合对Java有所了解的程序员，无须任何基础的数学理论或是函数式编程经验即可快速上手！

Original English Language edition published by Manning Publications, USA. Copyright © 2017 by Manning Publications. Simplified Chinese-language edition copyright © 2018 by Publishing House of Electronics Industry. All rights reserved.

本书简体中文版专有出版权由Manning Publications 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2017-2241

图书在版编目（CIP）数据

Java函数式编程/（法）皮埃尔·伊夫斯·索蒙特（Pierre-Yves Saumont）著；高清华译. —北京：电子工业出版社，2018.1

书名原文：Functional Programming in Java

ISBN 978-7-121-33021-6

I. ①J… II. ①皮… ②高… III. ①JAVA语言—程序设计 IV. ①TP312.8

中国版本图书馆CIP数据核字（2017）第277090号

策划编辑：张春雨

责任编辑：刘 舫

印 刷：北京京科印刷有限公司

装 订：北京京科印刷有限公司

出版发行：电子工业出版社

北京市海淀区万寿路173信箱

邮编：100036

开 本：787×980 1/16

印张：32.25 字数：578千字

版 次：2018年1月第1版

印 次：2018年1月第1次印刷

定 价：119.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

译者序

有幸受邀翻译本书。初见书名，心中不免有几分疑虑，难道又是一本教授如何使用 Java 8 lambda 来进行函数式编程的书吗？翻了几页，方觉自己大误。本书其实意在如何从零开始，逐步理清函数式编程的思维方式并编写基础类库，不仅授之以鱼，而且授之以渔。只不过由于 Java 的受众实在太广，所以才使用这门语言罢了。

函数式编程有一个至关重要的前提，那就是函数的输出只能取决于函数的参数（我们会在书中看到生成随机数的例子）。初看上去似乎与 Java 这门面向对象的语言不搭，但语言只是工具而已，正如你也可以在 Haskell 中编写命令式风格的代码一样。在一个不太复杂甚至非并发的常规 Java 系统中，由于程序内部状态的改变，多次调用同一个方法的返回值很可能是不一样的，更不用说所带来的副作用了。在函数式编程中，确定的输入决定了确定的输出，这意味着只要参数对了，结果一定在预期中。也就是说，函数式编程没有无法重现的 bug。在这样的前提下，单元测试相对容易实现，而且能极大地增强你的信心。（想想你对目前所在项目的单元测试有多大的信心？）许多个这样的函数复合起来，在不改变信心的同时能够提供更多、更强大的功能，进而带来更大的收益，如无状态的线程安全、必要时才计算的惰性求值、加快多次执行速度的记忆化等。

传统的命令式编程是计算机硬件的抽象，源自图灵机，其实就是外部输入、内部状态、对外部的输出以及对内部状态的改变。函数式编程源自 λ 演算，即将变

量和函数替换为值或表达式并根据运算符进行计算。函数式编程相比命令式编程，代码更简洁、可读性更强，这是因为它的思维方式更倾向于描述什么，而不是怎么做。所以学习过程反而更加自然，并且不需要多么高深的数学基础。可是我们也知道，软件开发没有银弹。新的方法论也会带来新的问题，需要运用新知识来解决。幸运的是，新知识的坑已经有人帮你踩过了，高阶函数、偏应用函数、复合函数、柯里化、闭包……软件开发从来不缺术语。幸好它们并非高不可攀，作者在第2章中帮你扫清疑虑，并在后续章节中挑战惰性求值、记忆化、状态处理、应用作用还有 actor 等更高级的技术。你说 Monad？作者才不告诉你它究竟是什么，但是看完本书你自然就领悟了。

函数式编程不是万能药。它有自己擅长的领域，也有自己的弱项。函数式编程是级别更高的抽象。高级别抽象带来的收益就是易读、好写，可是有些低级别的事情（如果你真的需要的话）可能就不容易完成。函数式编程没有副作用，导致无法完成输入/输出操作。尽管如此，你也会在本书中看到一些解决办法。函数式编程没有变量，因此无法改变循环的终止条件，故而没有循环，严重依赖于用递归来抽象循环。在某些情况下可能会影响性能，所以你还会在本书中看到一些性能与情怀之间的权衡。绝大部分的编程最佳实践都是针对某个特定场景而言的，因此脱离业务场景直接讨论技术并不可取。拥有函数式编程的思维，你就拥有了解决问题的另一种选择，但是条条大路通罗马，千万别钻牛角尖。程序是对现实世界的建模，“不要让世界适应你的模型，要让你的模型适应世界。”

高清华

译者简介

高清华

亚马逊软件研发工程师。工作十多年来，在简洁代码、自动化测试、持续交付、DevOps 等方面都有着丰富的经验。他是《DevOps 实践》译者之一。其技术博客的网址是 <http://qinghua.github.io/>，希望能以通俗易懂的语言普及 IT 技术。

为什么要函数式编程

函数式编程中没有赋值语句，因此变量一旦有了值，就不会再改变了。更通俗地说，函数式编程完全没有副作用。除了计算结果，调用函数没有其他作用。这样便消除了 bug 的一个主要来源，也使得执行顺序变得无关紧要——因为没有能够改变表达式值的副作用，可以在任何时候对它求值。这样便把程序员从处理控制流程的负担中解脱出来。由于能够在任何时候对表达式求值，所以可以用变量的值来自由替换表达式，反之亦然——程序是“引用透明”的。这样的自由度让函数式的程序比它们的一般对手在数学上更易驾驭。

—John Hughes

Why Functional Programming Matters

我称之为十亿美元的错误……我当时的目标是确保所有引用的使用都应该绝对安全，由编译器自动执行检查。但是我无法抵制引入一个空引用的诱惑，仅仅是因为它很容易实现。这导致了无数的错误、漏洞和系统崩溃，它很可能在过去的四十年中造成了十亿美元的痛苦和损害。

—Tony Hoare

测试程序可以是一个证明 bug 存在的有效方式，但是对于证明 bug 不存在还是无可奈何。

—Edsger W. Dijkstra

测试本身并不能提高软件质量。测试结果是质量的指标，但它们并不能提高质量。尝试通过增加测试来提高软件质量就像是尝试通过经常称自己的体重来减肥一样。

—Steve McConnell

注释的合理运用是为了补偿我们无法成功地在代码中表达自己。

—Robert C. Martin

编程的难点并不是解决问题，而是决定要解决什么问题。

—Paul Graham

面向对象编程通过封装变化的部分使代码容易理解。函数式编程通过最小化变化的部分使代码容易理解。

—Michael Feathers

我总是发现计划没有用，但是制订计划不可或缺。

—Dwight D. Eisenhower

设计软件有两种方法：一种是简单化，使其明显没有缺陷；另一种是复杂化，使其没有明显缺陷。前者要难得多。

—Tony Hoare

如果我们询问客户他们想要什么，他们只会说：“更快的马。”

—Henry Ford

尽管一些声明式程序员对等式推理 (equational reasoning) 只会耍嘴皮子，但函数式语言的用户每次运行编译器时都会用到它们，无论他们是否注意到。

—Philip Wadler

How to declare an imperative

我们并不试图争取 Lisp 程序员；我们在追赶 C++ 程序员。我们设法在他们到达 Lisp 的半路上拽走他们。

—Guy Steele

人们“得到”类型，一直用着它们。告诉某人他不能用香蕉来敲钉子并不令其感到惊讶。

—Unknown

TDD 代替了类型检查器……正如烈酒代替了悲伤。

—byorgey

首先，调试的难度是编写代码的两倍。因此，即使你把代码写得精彩绝伦，根据定义，你还不足以聪明地调试代码。

—Brian W. Kernighan 和 P. J. Plauger

一旦开始编程，令我们感到惊讶的是，很难让程序按预想的方案正确地工作。不得不靠调试来弄明白。我还记得当意识到我此生的大部分时间都会用于寻找自己程序中的错误时的那一刻。

—Maurice Wilkes (1949)

序言

编程序既有趣又多金。许多人为了乐趣而编程，还能赚钱。从这种意义上说，程序员有点像演员、音乐家或职业足球运动员。似乎是一个梦想，直到作为程序员的你开始负起真正的责任。从这个角度上说，编写游戏或办公应用程序都没有什么大不了的。如果应用程序有一个 bug，你只修复它并发布一个新版本即可。但如果你编写的程序被人们依赖，并且还不能简单地发布一个新版本后让用户们自行安装，那就是另一种情况了。当然，Java 并非用于编写监控核电站或控制飞机的应用程序，或者是一个简单的 bug 就可能置人类生命于风险中的系统。但如果你的程序用于管理互联网骨干，你一定不会愿意在奥运会开幕前一天发现一个讨厌的 bug，导致整个国家的电视传输失败。对于这样的程序，你希望确保它可以被证明是正确的。

大多数命令式程序无法被证明是正确的。测试只允许我们在测试失败时证明程序不正确。成功的测试说明不了什么问题。你无法证明发布的程序是不正确的。就单线程程序而言，大量的测试也许能够说明你的代码大部分是正确的。但是对于多线程应用程序，条件组合的数量使测试成为不可能。显然，我们需要另一种不同的方式来编写程序。在理想情况下，这种方法将允许我们证明程序是正确的。因为这不是一般不是完全可能的，所以一个很好的折中是明确分离程序中可以被证明为正确的部分和不能证明为正确的部分。这就是函数式编程技术所能提供的。

函数式编程的定义与函数式程序员一样多。有人说函数式编程是用函数编程。

这是对的，但它无助于你了解这种编程范式的优势。更重要的是，函数式编程需要将抽象推至极致的理念。它允许明确分离程序中可以被证明为正确的部分与输出取决于外部条件的其他部分。通过这种方式，函数式程序是不太容易产生 bug 的程序，它的 bug 只会驻留在特定的受限区域。

可以采用许多种技术来实现这一目标。使用不可变数据虽然不仅限于函数式编程，但却是这样一种技术。如果数据不能更改，你将不会有任何（不良的）意外，数据不会过期或损坏，没有竞争条件，无须锁住并发访问，也不会有死锁的风险。可以毫无风险地共享不可变数据。你不需要生成防御性副本，那就没有了忘记这样做的风险。另一种技术是抽象控制结构，因此你不必冒着混淆循环索引和退出条件的风险一次又一次地编写相同的结构。完全删除使用 null 引用（无论是隐式还是显式）将把你从臭名昭著的 NPE（空指针异常，NullPointerException）中解放出来。通过所有这些技术（还有更多），你可以确信：如果程序通过编译，那它就是正确的（也就是说，它的实现没有 bug）。虽然这样做并不能消除所有可能的 bug，但它更加安全。

计算机从一开始就基于寄存器中的可变值使用了命令式范式。正如其他许多被称为“命令式语言”的编程语言一样，Java 似乎在很大程度上依赖于这种范式，但根本没有必要。如果你是有经验的 Java 程序员，你可能会惊讶地发现无须更改变量的值就可以编写有用的程序。这并非函数式编程的必要条件，但是函数式程序员几乎总是自由自在地用着不可变数据。你可能也难以相信，可以在不使用 if...else 结构、while 还有 for 循环的情况下编写程序。再说一次，避免这样的结构并不是使用函数式范式的一个条件，但是如果你愿意，就可以避免这种结构，从而得到更安全的程序。所以即使 Java 通常被视为“命令式语言”，但它其实不是。没有命令式语言，也没有函数式语言。相信它们的存在就像是认为英语更适用于商业文档，而意大利语更适用于歌剧，法语更适用于情诗，德语更适用于哲学（或你能想象的任意组合）。差异可能存在，但它们大多是文化方面的，编程语言也是如此。Java 是一门命令式语言，这是因为大多数 Java 程序员是命令式程序员，而 Java 的文化也是命令式的。与此相反，Haskell 程序通常以函数式风格编写，因为程序员们都想要选择这门语言来进行函数式编程。但是可以在 Haskell 中编写命令式程序，也可以用 Java 编写函数式程序。不同之处在于，Haskell 比 Java 更加“对函数式友好”。

所以问题是：“你应该用 Java 来进行函数式编程吗？”令人惊讶的是（在给定本书主题的情况下），答案为否。要是可以自由地选择任何语言，我会建议你不应该为

这个目的而选择 Java。但是你通常没有这样的自由。在撰写关于使用 Java 进行函数式编程的文章时，我收到的大多数负面评论都是“你不应该为此使用 Java。这样用不是 Java 的本意。”或者“为什么要用 Java，用 Haskell、Scala 或其他什么不是更好？”

在现实中，你通常无法选择语言。如果你在一个公司工作，可能必须使用公司的语言，或者至少是你所在的团队为你正在工作的项目选择的语言。从这个角度上看，Haskell 从来都不是一个可选项。通常，除了 Java，你别无选择。如果你处于选择语言的位置，除了使用自己熟悉的语言或是使用允许重用一些旧代码或适合环境或是其他一些条件的语言之外，可能也别无选择。本书面向的正是你，除了使用 Java 之外别无选择的 Java 程序员，尽管你希望能受益于函数式编程的安全性。

在 Java 中使用函数式编程技术往往会导致你反对所谓的“最佳实践”。实际上，这些实践中有许多都没有用，有些还确实是非常糟糕的做法。从不捕获错误就是其中之一。作为 Java 程序员，你可能已经了解到不应该捕获 OOME（内存不足的错误，Out Of Memory Error）或是你无法处理的其他类型错误。也许你甚至知道不应该捕获 NPE（空指针异常，NullPointerExceptions），因为它们表示有 bug，你应该让应用程序崩溃并修复它。不幸的是，OOME 和 NPE 都不会使应用程序崩溃。它们只会使所在的线程崩溃，并留下处于某种不确定状态的应用程序。即使它们发生在主线程中，如果某些非后台线程正在运行，它们也可能无法使应用程序崩溃。当所有的应用程序都是单线程时，这种“最佳实践”是正确的。而现在它是一个非常糟糕的实践。你应该捕获所有的异常，尽管可能不在一个 try...catch 块中。函数式编程的真言是“始终捕获，绝不抛出”。

在我们的函数式编程之旅中还将挑战其他许多最佳实践。其中一个：“不要重复发明轮子”，尽管它与 Java 或命令式编程没有直接关系。想想看。某次，有人发明了轮子。那时候，大概是用刚性材料制造并随轴转动的圆。从那时起，轮子已经被重复发明许多次了。如果没有，那就不会有汽车，不会有火车，几乎没有什么东西会使用轮子。所以你应该继续努力一次次地重复发明轮子。不仅在将来会给我们带来更好的轮子，而且它具有挑战性，有回报，还有乐趣。（如果你认为当今的汽车轮子是圆形的，那么最好再思考一下，没有车能够开在圆形的轮子上！）¹

1 车轮与地面接触的部分是平的。作者以此为例，意在表达我们认为理所当然的事情并不一定是正确的。——译者注

致 谢

我要感谢许多参与制作本书的人。

首先，非常感谢我的策划编辑 Marina Michaels。非常荣幸与你一起工作，此外，还有你在手稿上的出色工作。

也非常感谢我的技术编辑 Mark Elston 和我的技术审校 Alessandro Campeis，你们帮我把这本书制作得比我自己弄要好得多。

还要谢谢所有的评论者、MEAP 的读者以及其他所有提供反馈和评论的人！如果没有你们的帮助，这本书不会是今天这个样子。特别的，我要感谢所有花时间审校和评论这本书的人：Aditya Kumar、Al Krinker、Andy Kirsch、Andy Knight、Anthony Moralez、Arun Allamsetty、Barry Kern、Boris Vasile、Bruce Hernandez、Charles Feduke、Chris Kirk、David Drummond、Davide Fiorentino lo Regio、Erwin van Eijk、Gualtiero Testa、Ivan Milosavljević、Jan Vorwerk、Jérôme Baton、Joshua McAdams、Julian Templeman、Maria Gemini、Norbert Kuchenmeister、Philippe Charrière、Piotr Bzdyl、Rambabu Posa、Sebastian Hähnel、Sebastian Metzger、Simeon Leyzerzon、Tarin Gamberini、Ursin Stauss、William Wheeler、Zach Schwartz 和 Zorodzayi Mukuya。

关于本书

这不是一本关于 Java 的书。这本书是关于函数式编程的，它是讲解编写软件程序的不同方法的。“不同”意味着它与称为“命令式范式”的编写软件的“传统”方式不同。本书涉及应用函数式范式进行 Java 编程。

没有像“函数式语言”这样的东西，只有或多或少对函数式友好的语言。虽然我在本书中使用 Java，但你可以将我所教的所有原则应用于任何其他语言。只是实现这些原则的方式有所不同。你可以用任何语言编写函数式程序，甚至是那些据说根本不是函数式的语言；同样，你也可以使用对函数式最友好的语言来编写命令式程序。

随着 Java 8 的发布，一些函数式特性已被添加到了 Java 语言中。但是正如本书不是关于 Java 的，它也不是关于这些具体的 Java 8 特性的。在这本书中，我大量地使用了某些特性，并几乎忽略了其他特性。如果你的目标是学习如何使用 Java 8 的函数式特性，本书并不合适。Urma、Fusco 和 Mycroft 编写的 *Java 8 in Action* (Manning, 2014) 会是一个更好的选择。

另一方面，如果你想了解什么是函数式编程，如何构建函数式数据结构，以及函数式编程范式如何帮助你编写更好的程序（有时使用 Java 8 的特性，有时又避免使用它们），那么这就是为你量身定做的书。

本书面向的读者

本书面向具有一些 Java 编程经验的读者，需要对 Java 泛型有较好的理解。如果你发现自己不了解 Java 的构造（例如实现为方法的泛型常量或者是参数化方法调用），请不必担心：我将解释它们的含义以及为什么需要它们。

你不必具备函数式编程的经验，或是了解基础的数学理论。第 2 章将会让你回忆起什么是一个函数。不会再用其他的数学知识了。

我介绍的所有函数式技术都与它们相应的命令式相关，所以我希望你具有在 Java 中使用命令式编程的经验。

如何使用本书

希望读者能顺序阅读本书，因为每一章都建立在前文概念的基础之上。第 14 章和第 15 章是仅有的例外，不会使用你在第 12 章和第 13 章中学习的内容。也就是说，如果你愿意，可以跳过第 12 章和第 13 章；它们提供了更先进的技术，理解了会很有用，但是你可能不会在自己的程序中使用。

我用了“阅读”这个词，但这本书并不只是为了阅读而生，它只有极少部分的理论。为了充分利用这本书，请在电脑前阅读，随时动手做练习。每一章都包含了一些练习，并提供了必要的说明和提示，以帮助你找到答案。所有的代码都可以从 GitHub (<http://github.com/fpinjava/fpinjava>) 和出版商的网站 <https://www.manning.com/books/functional-programming-in-java> 上免费下载。每个练习都附带一个推荐答案和 JUnit 测试，你可以用它来验证你的答案是否正确。

为了导入 IntelliJ（推荐）、NetBeans 或 Eclipse 中，代码附带了所有必需的要素，虽然在撰写本文时，Eclipse（Mars 4.5.1）尚未完全兼容 Java 8。可以通过源代码或者使用 Gradle 导入项目。Gradle 的任何版本都可以，因为它能够自动下载正确的版本。

请注意，我并不希望你仅靠阅读文章就能够理解本书中出现的大部分概念。做练习可能是学习过程中最重要的部分，所以我鼓励你不要跳过任何练习。有些练习可能做起来很困难，而你可能会忍不住想看看推荐答案。这样做完全没问题，但是你应该在那之后回到练习中，不看答案再做一次。如果只是翻答案，你可能会在之后尝试解决更高级的练习时再次遇到问题。

这种方式不需要烦琐地打字，因为几乎没有什么要复制的。大部分练习由编写

方法的实现组成，我为此提供了环境和方法签名。没有多于十几行代码的练习；大多数都是四五行。

一旦你完成了练习（即你的实现通过编译），只需运行相应的测试来验证它是否正确即可。

值得注意的是，每一章中的各个练习都是独立的，所以在同一章中，上一个练习创建的代码会复制到下一个练习中。这种做法的必要性在于，因为每个练习通常会建立在前面的练习基础之上，因此虽然可以使用相同的类，但是实现不同。所以，不要在完成前面的练习之前先看后面的练习，因为你会看到还未做的练习的答案。

你可以下载代码的存档文件，也可以用 `Git clone`。我强烈建议用 `clone`，因为代码可能会更改，通过简单的 `pull` 命令来更新代码比重新下载完整的存档文件效率更高。

练习的代码都被分到了各个模块中，它们的名字与章节标题而不是章节号大体上相对应。因此，IDE 将按字母顺序排列它们，而不会按照它们在书中出现的顺序排序。为了帮助你了解每个章节对应的模块，我还在随书附带代码 (<http://github.com/fpinjava/fpinjava>) 的 README 文件中提供了一个包含相应模块名称的章节列表。

设定期望

函数式编程并不会比命令式编程更难，只是不同而已。你可以用这两个范式来解决相同的问题，但是从一个转换到另一个时可能效率较低。学习函数式编程就像是学习一门外语。正如你想着一门语言并翻译到另一门语言不可能效率较高那样，你也不能想着命令式并把代码翻译为函数式。并且正如你需要学会用新的语言来思考那样，你也需要学会函数式地思考。仅靠阅读不足以学习函数式地思考，编写代码必不可少。所以你必须多加练习。

这就是为什么我不希望你只是通过阅读来了解这本书，为什么我要提供这么多练习？你需要做练习，以完全掌握函数式编程的概念。并不是因为这个主题很复杂，不可能通过阅读来理解它，而是因为如果你只通过阅读（不做练习）就能理解它，那你很可能不需要这本书。

出于所有的这些原因，练习是充分利用本书的关键。我鼓励你在继续阅读之前尝试解决每个练习。如果你找不到答案，请再试一试，而不是直接翻阅我提供的答案。

如果你理解一些东西很困难，请在论坛上提问（见下一小节）。在论坛上提出问题和获得答案不仅可以帮助你，还可以帮助回答问题的人（以及其他有同样问题的人）。我们都通过回答问题（顺便说一下，主要是我们自己的问题）来学习，而较少通过问问题的方式。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在[提交勘误](#)处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **与我们交流**：在页面下方[读者评论](#)处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/33021>



目录

第1章 什么是函数式编程.....	1
1.1 函数式编程是什么	2
1.2 编写没有副作用的程序	4
1.3 引用透明如何让程序更安全	6
1.4 函数式编程的优势	7
1.5 用代换模型来推断程序	8
1.6 将函数式原则应用于一个简单的例子	9
1.7 抽象到极致	15
1.8 总结	16
第2章 在Java中使用函数	17
2.1 什么是函数	18
2.1.1 现实世界里的函数	18
2.2 Java中的函数	24
2.2.1 函数式的方法	24
2.2.2 Java 的函数式接口与匿名类	30
2.2.3 复合函数	31

2.2.4	多态函数	32
2.2.5	通过 lambda 简化代码.....	33
2.3	高级函数特性	36
2.3.1	多参函数怎么样	36
2.3.2	应用柯里化函数	37
2.3.3	高阶函数	38
2.3.4	多态高阶函数	39
2.3.5	使用匿名函数	43
2.3.6	局部函数	45
2.3.7	闭包	46
2.3.8	部分函数应用和自动柯里化	48
2.3.9	交换部分应用函数的参数	53
2.3.10	递归函数	54
2.3.11	恒等函数.....	56
2.4	Java 8 的函数式接口	58
2.5	调试 lambda.....	59
2.6	总结	62
第3章	让Java更加函数式	63
3.1	使标准控制结构具有函数式风格	64
3.2	抽象控制结构	65
3.2.1	清理代码	69
3.2.2	if ... else 的另一种方式.....	73
3.3	抽象迭代	78
3.3.1	使用映射抽象列表操作	79
3.3.2	创建列表	80
3.3.3	使用 head 和 tail 操作.....	81
3.3.4	函数式地添加列表元素	83
3.3.5	化简和折叠列表	83
3.3.6	复合映射和映射复合	90
3.3.7	对列表应用作用	91
3.3.8	处理函数式的输出	92

3.3.9 构建反递归列表	93
3.4 使用正确的类型	97
3.4.1 标准类型的问题	97
3.4.2 定义值类型	99
3.4.3 值类型的未来	103
3.5 总结	103
第4章 递归、反递归和记忆化	104
4.1 理解反递归和递归	105
4.1.1 探讨反递归和递归的加法例子	105
4.1.2 在 Java 中实现递归	106
4.1.3 使用尾调用消除	107
4.1.4 使用尾递归方法和函数	107
4.1.5 抽象递归	108
4.1.6 为基于栈的递归方法使用一个直接替代品	112
4.2 使用递归函数	115
4.2.1 使用局部定义的函数	115
4.2.2 使函数成为尾递归	116
4.2.3 双递归函数：斐波那契数列示例	117
4.2.4 让列表的方法变成栈安全的递归	120
4.3 复合大量函数	123
4.4 使用记忆化	127
4.4.1 命令式编程中的记忆化	127
4.4.2 递归函数的记忆化	128
4.4.3 自动记忆化	130
4.5 总结	136
第5章 用列表处理数据	138
5.1 如何对数据集进行分类	138
5.1.1 不同的列表类型	139
5.1.2 对列表性能的相对期望	140
5.1.3 时间与空间，时间与复杂度的取舍	141
5.1.4 直接修改	142

5.1.5 持久化数据结构	143
5.2 一个不可变、持久化的单链表实现	144
5.3 在列表操作中共享数据	148
5.3.1 更多列表操作	150
5.4 使用高阶函数递归折叠列表	155
5.4.1 基于堆的 foldRight 递归版	162
5.4.2 映射和过滤列表	164
5.5 总结	167
第6章 处理可选数据	168
6.1 空指针的问题	169
6.2 空引用的替代方案	171
6.3 Option数据类型	174
6.3.1 从 Option 中取值	176
6.3.2 将函数应用于可选值	178
6.3.3 复合 Option 处理	179
6.3.4 Option 的用例	181
6.3.5 复合 Option 的其他方法	186
6.3.6 复合 Option 和 List	189
6.4 Option 的其他实用程序	191
6.4.1 检查是 Some 还是 None	191
6.4.2 equals 和 hashCode	192
6.5 如何及何时使用 Option	193
6.6 总结	195
第7章 处理错误和异常	197
7.1 待解决的问题	197
7.2 Either类型	199
7.2.1 复合 Either	200
7.3 Result类型	203
7.3.1 为 Result 类添加方法	204
7.4 Result模式	206
7.5 Result处理进阶	213

7.5.1	应用断言	214
7.5.2	映射 Failure	215
7.5.3	增加工厂方法	218
7.5.4	应用作用	220
7.5.5	Result 复合进阶	222
7.6	总结	226
第8章	列表处理进阶	228
8.1	length的问题	229
8.1.1	性能问题	229
8.1.2	记忆化的优点	230
8.1.3	记忆化的缺点	230
8.1.4	实际性能	232
8.2	复合List和Result	233
8.2.1	List 中返回 Result 的方法	233
8.2.2	将 List<Result> 转换为 Result<List>	235
8.3	抽象常见列表用例	238
8.3.1	压缩和解压缩列表	238
8.3.2	通过索引访问元素	241
8.3.3	拆分列表	244
8.3.4	搜索子列表	248
8.3.5	使用列表的其他函数	249
8.4	自动并行处理列表	254
8.4.1	并非所有的计算都可以并行化	254
8.4.2	将列表拆分为子列表	254
8.4.3	并行处理子列表	256
8.5	总结	258
第9章	使用惰性	259
9.1	理解严格和惰性	259
9.1.1	Java 是一门严格的语言	260
9.1.2	严格带来的问题	261
9.2	实现惰性	263

9.3	只有惰性才能做到的事	264
9.4	为何不要用Java 8中的Stream	265
9.5	创建一个惰性列表数据结构	266
9.5.1	记忆已计算的值	268
9.5.2	对流的操作	271
9.6	惰性的真正本质	274
9.6.1	折叠流	277
9.7	处理无限流	282
9.8	避免null引用和可变字段	285
9.9	总结	287
第10章	用树进行更多数据处理	289
10.1	二叉树	290
10.1.1	平衡树和非平衡树	291
10.1.2	大小、高度和深度	291
10.1.3	叶树	292
10.1.4	有序二叉树或二叉搜索树	292
10.1.5	插入顺序	293
10.1.6	树的遍历顺序	294
10.2	实现二叉搜索树	297
10.3	从树中删除元素	303
10.4	合并任意树	304
10.5	折叠树	310
10.5.1	用两个函数折叠	311
10.5.2	用一个函数折叠	313
10.5.3	选择哪种折叠的实现	314
10.6	映射树	316
10.7	平衡树	317
10.7.1	旋转树	317
10.7.2	使用DSW算法平衡树	320
10.7.3	自动平衡树	322
10.7.4	解决正确的问题	323
10.8	总结	324

第11章 用高级树来解决真实问题.....	325
11.1 性能更好且栈安全的自平衡树.....	326
11.1.1 树的基本结构.....	326
11.1.2 往红黑树中插入元素.....	331
11.2 红黑树的用例: map.....	337
11.2.1 实现 map.....	337
11.2.2 扩展 map.....	340
11.2.3 使用键不可比较的 map.....	341
11.3 实现函数式优先队列.....	344
11.3.1 优先队列访问协议.....	344
11.3.2 优先队列使用案例.....	344
11.3.3 实现需求.....	345
11.3.4 左倾堆数据结构.....	345
11.3.5 实现左倾堆.....	346
11.3.6 实现像队列一样的接口.....	351
11.4 元素不可比较的优先队列.....	352
11.5 总结.....	358
第12章 用函数式的方式处理状态改变.....	359
12.1 一个函数式的随机数发生器.....	360
12.1.1 随机数发生器接口.....	361
12.1.2 实现随机数发生器.....	362
12.2 处理状态的通用API.....	366
12.2.1 使用状态操作.....	367
12.2.2 复合状态操作.....	368
12.2.3 递归状态操作.....	370
12.3 通用状态处理.....	372
12.3.1 状态模式.....	374
12.3.2 构建一个状态机.....	375
12.3.3 何时使用状态和状态机.....	381
12.4 总结.....	381

第13章 函数式输入/输出	382
13.1 在上下文中应用作用	383
13.1.1 作用是什么	384
13.1.2 实现作用	384
13.1.3 用于失败情况的更强大的作用	387
13.2 读取数据	390
13.2.1 从控制台读取	390
13.2.2 从文件中读取	395
13.2.3 检查输入	396
13.3 真正的函数式输入/输出	398
13.3.1 怎样才能让输入/输出是完全函数式的	398
13.3.2 实现纯函数式的输入/输出	399
13.3.3 合并 IO	400
13.3.4 用 IO 处理输入	402
13.3.5 扩展 IO 类型	404
13.3.6 使 IO 类型栈安全	407
13.4 总结	413
第14章 通过actor共享可变状态	414
14.1 actor模型	415
14.1.1 异步消息	416
14.1.2 处理并行	416
14.1.3 处理 actor 状态变化	417
14.2 构建actor框架	418
14.2.1 actor 框架的限制	418
14.2.2 设计 actor 框架接口	418
14.2.3 AbstractActor 的实现	420
14.3 开始使用actor	422
14.3.1 实现乒乓示例	422
14.3.2 一个更严谨的例子：并行运行一个计算	424
14.3.3 重新排序结果	430
14.3.4 解决性能问题	433
14.4 总结	439

第15章 以函数式的方式解决常见问题	440
15.1 使用断言来校验数据	441
15.2 从文件中读取属性	446
15.2.1 载入属性文件	446
15.2.2 将属性读取为字符串	447
15.2.3 生成更好的错误消息	448
15.2.4 像列表那样读取属性	451
15.2.5 读取枚举值	453
15.2.6 读取任意类型的属性	454
15.3 转换命令式程序：XML读取器	457
15.3.1 列出必需的函数	459
15.3.2 复合函数并应用作用	460
15.3.3 实现函数	461
15.3.4 让程序更加函数式	462
15.3.5 修复参数类型问题	466
15.3.6 以处理元素的函数为参数	467
15.3.7 处理元素名称错误	468
15.4 总结	470
附录A 使用Java 8的函数式特性	471
附录B Monad	479
附录C 敢问路在何方	485

什么是函数式编程

本章要点

- 函数式编程的优势
- 副作用的问题
- 引用透明如何让程序更安全
- 使用代换模型编程的原因
- 充分利用抽象

并不是所有人都在函数式编程（functional programming，即 FP）的定义上达成了共识。一般来说，函数式编程是用函数来编程的一种编程范式。但这个定义并不能解释最重要的一点：函数式编程和其他编程范式的区别，以及究竟是什么让它（可能）成为编程的更佳方式。在 1990 年出版的 *Why Functional Programming Matters* 一书中，John Hughes 写道：

函数式编程中没有赋值语句，因此变量一旦有了值，就不会再改变了。更通俗地说，函数式编程完全没有副作用。除了计算结果，调用函数没有别的作用。这样便消除了 bug 的一个主要来源，也使得执行顺序变得无关紧要——因

为没有能够改变表达式值的副作用，可以在任何时候对它求值。这样便把程序员从处理控制流程的负担中解脱出来。由于能够在任何时候对表达式求值，所以可以用变量的值来自由替换表达式，反之亦然——即程序是“引用透明”的。这样的自由度让函数式的程序比它们的一般对手在数学上更易驾驭。¹

在本章的其余内容中，我会简要地展示引用透明和代换模型的概念，还有函数式编程的其他精华概念。你将会在其他章节中多次应用这些概念。

1.1 函数式编程是什么

理解事物是什么与不是什么往往都很重要。如果函数式编程是一种编程范式，那么显然还会有其他不同的编程范式。与一些人预料的可能相反，函数式编程与面向对象编程（OOP）并不是非此即彼。有些函数式编程语言是面向对象的，有些不是。

函数式编程有时被认为是一系列可以补充或替代其他编程范式的技术，例如

- 函数是一等公民
- 匿名函数
- 闭包
- 柯里化
- 惰性求值
- 参数多态
- 代数数据类型

尽管大部分的函数式编程语言确实使用了一些这样的技术，但是对于每一种技术，你也都可以找到函数式编程语言不支持的例子。同样的，非函数式编程语言也会支持一些这样的技术。你会看到在本书中学习这些技术的时候，让程序更加函数化的并非是编程语言，而是你写代码的方式。话虽如此，有些语言会对函数更加友好一些。

与函数式编程相对的应该算是命令式编程范式。在命令式编程的风格里，程序

¹ John Hughes, “Why Functional Programming Matters,” from D. Turner, ed., *Research Topics in Functional Programming* (Addison-Wesley, 1990), 17–42, www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf.

由“做”事情的要素构成。“做”事情意味着一个初始状态、一个转换过程和一个终止状态。有时这被称为状态改变 (state mutation)。传统的命令式风格的程序通常描述了一系列由条件判断区分的改变。例如，正数 a^1 和 b 相加的程序，可以表示为如下伪代码：

- 如果 $b == 0$ ，返回 a 。
- 否则 a 自增，并且 b 自减。
- 用新的 a 和 b 重新计算。

在这段伪代码里，你可以识别出大多数命令式编程的传统指令：测试条件、可变变量、分支以及一个返回值。这段代码可以表示为如图 1.1 所示的流程图。

另一方面，函数式编程由“是”什么的元素组成，而不是“做”什么。 a 与 b 的和并不会“造”出一个结果。例如 2 与 3 的和，并不会造出 5。它就是 5。

这个差异似乎不那么重要，但并非如此。最主要的结果是，每当你遇到 $2+3$ 时，就可以把它替换为 5。你能在命令式编程中同样这么

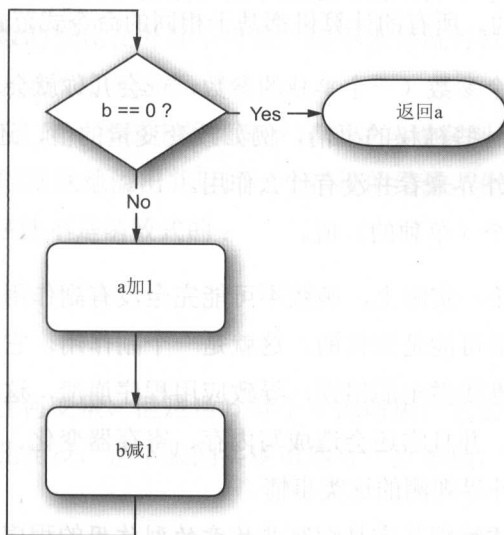


图 1.1 展示了发生在一段时间内的命令式程序的流程图。在得到结果之前，许多东西都改变了，状态也改变了。

做吗？好吧，有时候可以。但是有时候不改变程序的结果就无法做到。如果你打算替换掉的表达式除了返回结果以外没有其他作用，那就可以把它安全地替换为结果。但是你怎么知道它有没有其他作用呢？在求和的例子里，你可以清楚地看到 a 与 b 这两个变量被程序破坏掉了。这就是程序除了返回一个结果以外的作用，因而被称为副作用。（发生在 Java 方法中的计算还不太一样，因为变量 a 和 b 可能是传进来的值，仅在当前作用域里发生变化，在方法外面并不可见。）

¹ 此书英文原书中使用的公式、函数、变量等书写形式与我国国标中的标准有所不同，但因本书中涉及的相关内容较多，故沿用原书中的写法。——译者注

命令式编程和函数式编程的一个最大的不同是，函数式编程没有副作用。这意味着其中

- 没有变量改变。
- 没有打印到控制台或其他设备。
- 没有写入文件、数据库、网络或其他什么。
- 没有抛出异常。

当我说“没有副作用”的时候，我是指没有可观测到的副作用。函数式的程序是由接收参数并返回值的函数复合而成的，仅此而已。你并不关心函数内部发生了什么，因为在理论上，什么都没有发生。但是在实际上，程序是为完全不函数式的计算机而编写的。所有的计算机都基于相同的命令式范式，所以函数就是如下黑盒：

- 接收一个参数（一个单独的参数，一会儿你就会看到）。
- 内部做一些神秘的事情，例如改变变量的值，还有许多命令式风格的东西，但是在外界来看并没有什么作用。
- 返回一个（单独的）值。

这只是理论。实际上，函数不可能完全没有副作用。函数会在某个时候返回一个值，而这个值可能是变化的。这就是一个副作用。它可能会造成一个内存耗尽的错误，或者是堆栈溢出的错误，导致应用程序崩溃，这在某种意义上就是一个可观测到的副作用。并且它还会造成写内存、寄存器变化、加载线程、上下文切换和其他确实会影响外界观测的这类事情。

所以函数式编程其实是编写非故意的副作用的程序，我的意思是，副作用是程序预期结果的一部分。非故意的副作用也应该越少越好。

1.2 编写没有副作用的程序

你可能想知道如何才能编写出既没有副作用又有用的程序。显然这不太可能。函数式编程并非关于编写没有可观测结果的程序。它是关于编写除了返回值以外没有可观测结果的程序。但如果这就是程序能做的全部，那就没有多大的用处。最后，函数式编程需要有可观测的作用，例如把结果显示在屏幕上，写到文件或数据库里，或者通过网络发送出去。与外界的这种交互不会发生在计算过程中，而只会发生在

计算完成后。换句话说，将会推迟副作用并单独应用。

以图 1.1 的加法为例。虽然描述的是命令式的风格，但程序也可能是函数式的，取决于如何实现。想象一下用 Java 实现的以下代码：

```
public static int add(int a, int b) {  
    while (b > 0) {  
        a++;  
        b--;  
    }  
    return a;  
}
```

这段程序是彻头彻尾的函数式。它接收一对整型 a 和 b 为参，并返回一个值，完全没有其他可观测的作用。它改变了变量，但事实上与需求并不矛盾，因为 Java 的参数是值传递的，所以参数的变化在外界不可见。接下来你就可以选择应用一个作用，例如显示结果或是用结果做其他运算。

请注意，虽然结果可能不正确（在溢出的情况下），但是与没有副作用并不矛盾。如果 a 和 b 太大了，程序可能默默地溢出并返回一个错误的结果，但是这仍然是函数式的。另一方面，以下程序就不是函数式的：

```
public static int div(int a, int b) {  
    return a / b;  
}
```

虽然这段程序并不改变任何变量，但是当 b 等于 0 的时候，它会抛出异常。抛出异常就是一个副作用。与此相反，接下来的实现虽然有一点笨拙，但它却是函数式的：

```
public static int div(int a, int b) {  
    return (int) (a / (float) b);  
}
```

即使 b 为 0，这样的实现也不会抛出异常，但是它会返回一个特殊的值。由你自行决定你的函数用返回特殊值来代表除数为零的做法是否可行。（很可能不行！）

无论抛异常是有意为之或是无意的，它终归是一个副作用。尽管在命令式编程里，副作用一般正是我们想要的。最简单的形式可能看起来如下：

```
public static void add(int a, int b) {  
    while (b > 0) {  
        a++;  
        b--;  
    }
```

```
}  
System.out.println(a);  
}
```

这段程序并不返回值，而是把它打印到了控制台上。这就是期望的副作用。

请注意如下程序，既返回了值又有意加上了副作用：

```
public static int add(int a, int b) {  
    log(String.format("Adding %s and %s", a, b));  
    while (b > 0) {  
        a++;  
        b--;  
    }  
    log(String.format("Returning %s", a));  
    return a;  
}
```

由于日志的副作用，上面这段代码并不是函数式的。

1.3 引用透明如何让程序更安全

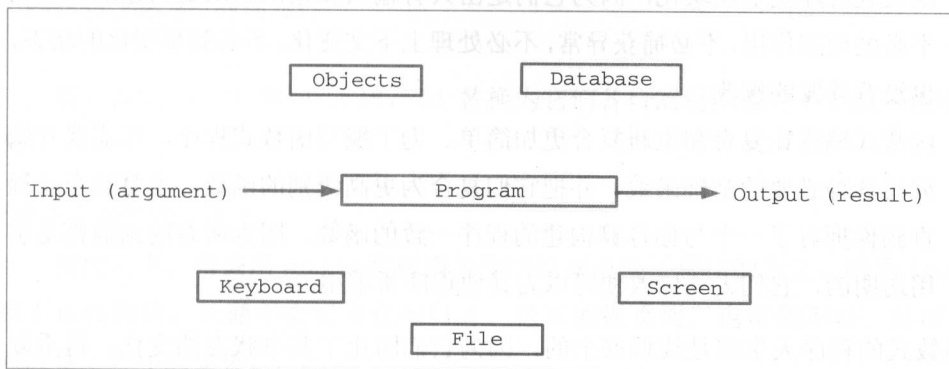
没有副作用(所以并不会改变外界的什么)并不足以让程序变成函数式的。同样，函数式编程也不能被外界所影响。换句话说，函数式程序的输出只能取决于自己的参数。这就意味着函数式代码不能从控制台、文件、远程 URL、数据库甚至是系统里读取数据。不被外界所影响的代码就是引用透明的。

引用透明的代码有一些性质对程序员而言很有意思：

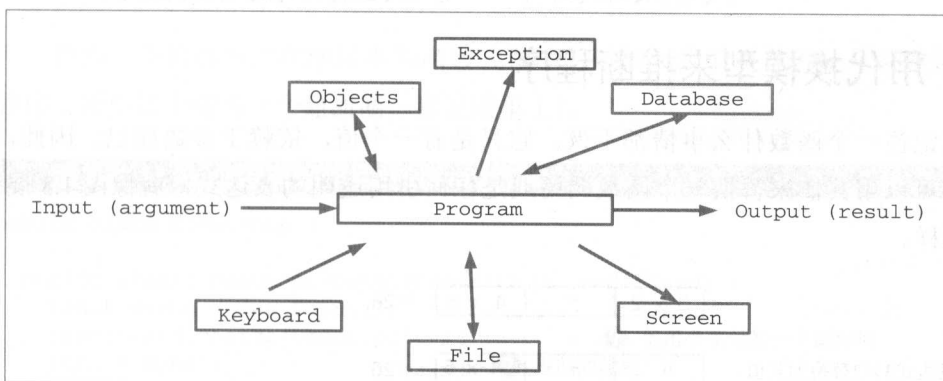
- 它是独立的。它并不依赖于任何外部的设备来工作。你可以在任何上下文中使用它——你需要做的一切就是提供一个有效的参数。
- 它是确定的，意味着相同的参数总是返回相同的结果。在引用透明的代码中，不会有意外发生。它可能返回一个错误的结果，但至少结果对于相同的参数而言是绝对不会变化的。
- 它绝对不会抛出任何种类的 Exception。它可能抛出错误，例如 OOME (out-of-memory error, 即内存耗尽) 或是 SOE (stack-overflow error, 即堆栈溢出)，但是这些错误表示代码有 bug，并不是作为程序员的你或是你 API 的用户应该处理的（除了让应用程序崩溃并最终修复 bug）。
- 任何时候它都不会导致其他代码意外失败。例如，它不会改变参数或是外界的数据，从而导致调用者发现自己的数据过期或者并发访问异常。

- 它不会由于外部设备（数据库、文件系统或网络）不可用、太慢或坏掉而崩溃。

图 1.2 展示了引用透明的程序和非引用透明的程序之间的区别。



一个引用透明的程序除了获取输入参数和输出结果以外，并不会影响外界。它的结果只是取决于参数。



一个非引用透明的程序可能会从外界读写数据、写日志、改变外界对象、读取键盘输入、输出到屏幕等。结果是不可预测的。

图 1.2 比较是否是引用透明的程序。

1.4 函数式编程的优势

从我刚刚说的那些，你应该可以猜到函数式编程的诸多优势：

- 函数式程序更加易于推断，因为它们是确定性（deterministic）的。对于一个特定的输入总会给出相同的输出。在许多情况下，你都可以证明程序是正确的，而不是在大量的测试后仍然不确定程序是否会在意外的情况下出错。

- 函数式程序更加易于测试。因为没有副作用，所以你不需那些经常用于在测试里隔离程序及外界的 mock。
- 函数式程序更加模块化，因为它们是由只有输入和输出的函数构建的。我们不必处理副作用，不必捕获异常，不必处理上下文变化，不必共享变化的状态，也没有并发的修改。
- 函数式编程让复合和重新复合更加简单。为了编写函数式程序，你需要开始编写各种必要的基础函数，并把它们复合为更高级别的函数，重复这个过程直到你拥有了一个与你打算构建的程序一致的函数。因为所有的函数都是引用透明的，它们无须修改便可以为其他程序所重用。

函数式的程序天生就是线程安全的，因为它们防止了共享状态的变化。再重复一遍，这并不意味着所有数据都需要不可变，只有共享的数据才需要。但是函数式程序员很快就会认识到不可变的数据总是更安全的，即使在外界观测不到这种变化。

1.5 用代换模型来推断程序

请记住一个函数什么事情都不做。它只是有一个值，依赖于参数而已。因此，永远都可以用其值来替换一个函数调用或是任何引用透明的表达式，就像图 1.3 展示的那样。

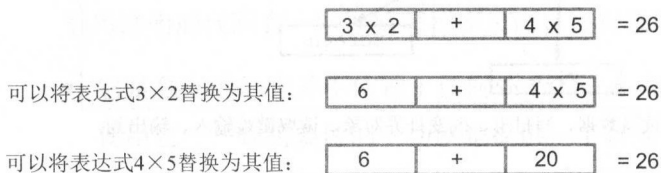


图 1.3 用值来替换一个引用透明的表达式并不会改变整体的含义。

当应用函数时，代换模型允许你将任何函数调用替换为它的返回值。思考如下代码：

```
public static void main(String[] args) {  
    int x = add(mult(2, 3), mult(4, 5));  
}  
public static int add(int a, int b) {  
    log(String.format("Returning %s as the result of %s + %s", a + b, a, b));  
    return a + b;  
}
```

```
public static int mult(int a, int b) {  
    return a * b;  
}  
public static void log(String m) {  
    System.out.println(m);  
}
```

将 `mult(2, 3)` 和 `mult(4, 5)` 替换为它们各自的返回值并不会改变程序的含义。

```
int x = add(6, 20);
```

相比之下，将调用 `add` 函数替换为返回值就改变了程序的含义，因为 `log` 方法将不再被调用，从而不会记录任何日志。这可能很重要，也可能不是。总而言之，它改变了程序的结果。

1.6 将函数式原则应用于一个简单的例子

作为一个将命令式程序转换为函数式程序的示例，我们将思考一个非常简单的程序：用信用卡购买一个甜甜圈，参见清单 1.1。

清单 1.1 一个带有副作用的 Java 程序

```
public class DonutShop {  
    public static Donut buyDonut(CreditCard creditCard) {  
        Donut donut = new Donut();  
        creditCard.charge(Donut.price);  
        return donut;  
    }  
}
```

← ① 信用卡支付是一个副作用
← ② 返回甜甜圈

在这段代码中，信用卡支付是一个副作用 ①。信用卡支付多半由调用银行、检查信用卡是否可用并已授权、注册交易等组成。函数返回甜甜圈 ②。

这种代码的问题在于难以测试。测试程序可能需要联系银行并用某个 `mock` 账户来注册交易。要不就得创建一张 `mock` 信用卡来代替真实的 `charge` 方法，并在测试之后验证 `mock` 的状态。

如果你想在无须接触银行或是使用 `mock` 的情况下测试代码，那就应该移除副作用。由于你仍然想要用信用卡支付，唯一的解决方案就是往返回值里加个什么东西来表示这个操作。你的 `buyDonut` 方法将会返回甜甜圈和表示支付的这个东西。

你可以使用一个 `Payment` 类来表示支付，如清单 1.2 所示。

清单 1.2 Payment 类

```
public class Payment {  
  
    public final CreditCard creditCard;  
    public final int amount;  
  
    public Payment(CreditCard creditCard, int amount) {  
        this.creditCard = creditCard;  
        this.amount = amount;  
    }  
}
```

这个类包含了表示支付的必要数据，由一张信用卡和支付金额组成。由于 buyDonut 方法需要返回 Donut 和 Payment 两个对象，你需要为此创建一个专门的类，如 Purchase：

```
public class Purchase {  
  
    public Donut donut;  
    public Payment payment;  
  
    public Purchase(Donut donut, Payment payment) {  
        this.donut = donut;  
        this.payment = payment;  
    }  
}
```

你经常会需要一个类来容纳两个（或以上的）值，因为函数式编程替换副作用的方式就是将其返回。

你可以使用被称为 Tuple（元组）的通用类，而不必创建一个特定的 Purchase 类。这个类将会由两种类型的参数组成（Donut 和 Payment）。清单 1.3 展示了它的实现，以及它在 DonutShop 类里的使用方法。

清单 1.3 Tuple 类

```
public class Tuple<T, U> {  
  
    public final T _1;  
    public final U _2;  
  
    public Tuple(T t, U u) {  
        this._1 = t;  
        this._2 = u;  
    }  
}
```

```
public class DonutShop {

    public static Tuple<Donut, Payment> buyDonut(CreditCard creditCard) {
        Donut donut = new Donut();
        Payment payment = new Payment(creditCard, Donut.price);
        return new Tuple<>(donut, payment);
    }
}
```

请注意，你不必顾虑（在这个时候）如何真正地用信用卡支付，这样可以给你构建程序带来一些自由。你仍然可以接着立即支付，也可以将它保存起来以便后续支付，甚至还可以将一张信用卡里保存的多份待支付记录合并起来，在一个操作里处理完成。这样便可以通过减少调用信用卡服务的次数来节省你的开销。

清单 1.4 里的 `combine` 方法允许你合并支付。请注意，如果信用卡不一致，将会抛出异常。这与我说的函数式编程不抛出异常并不矛盾。在这里，试图合并两张不同信用卡的两份待支付记录，被视为一个 **bug**，所以必须使应用程序崩溃。（这样做并不明智。你要到第 7 章之后才能学到如何用不抛出异常的方式来处理这种状况。）

清单 1.4 将多份待支付记录合并为一份

```
package com.fpinjava.introduction.listing01_04;

public class Payment {

    public final CreditCard creditCard;
    public final int amount;

    public Payment(CreditCard creditCard, int amount) {
        this.creditCard = creditCard;
        this.amount = amount;
    }

    public Payment combine(Payment payment) {
        if (creditCard.equals(payment.creditCard)) {
            return new Payment(creditCard, amount + payment.amount);
        } else {
            throw new IllegalStateException("Cards don't match.");
        }
    }
}
```

当然，用 `combine` 方法一次购买多份甜甜圈的效率并不高。在这种情况下，你可以用 `buyDonuts(int n, CreditCard creditCard)` 方法来简单地

代替 buyDonut 方法，如清单 1.5 所示。该方法返回一个 Tuple<List<Donut>, Payment>。

清单 1.5 一次购买多份甜甜圈

```
package com.fpinjava.introduction.listing01_05;

import static com.fpinjava.common.List.fill;
import com.fpinjava.common.List;
import com.fpinjava.common.Tuple;

public class DonutShop {

    public static Tuple<Donut, Payment> buyDonut(final CreditCard cCard) {
        return new Tuple<>>(new Donut(), new Payment(cCard, Donut.price));
    }

    public static Tuple<List<Donut>, Payment> buyDonuts(final int quantity,
                                                         final CreditCard cCard) {
        return new Tuple<>>(fill(quantity, () -> new Donut()),
                           new Payment(cCard, Donut.price * quantity));
    }
}
```

请注意，这个方法没有用标准的 java.util.List 类，因为这个类并没有提供你需要的一些函数式方法。在第 3 章中，你将会看到如何函数式地使用 java.util.List 类来编写一个小函数式库。然后在第 5 章，你将会开发一个全新的函数式 List，也就是这里所用的列表。这个 fill 方法与使用标准 Java 列表的以下代码有几分相似之处：

```
public static Tuple<List<Donut>, Payment> buyDonuts(final int quantity,
                                                      final CreditCard cCard) {
    return new Tuple<>>(Collections.nCopies(quantity, new Donut()),
                       new Payment(cCard, Donut.price * quantity));
}
```

由于很快将会需要额外的函数式方法，你就别用 Java 的列表了。目前你只需知道 static List<A> fill(int n, Supplier<A> s) 方法通过一个特殊的对象 Supplier<A> 创建了包含 n 个实例的一个集合。正如其名所示，一个 Supplier<A> 是一个对象，当其 get() 方法被调用时能够提供一个 A。用 Supplier<A> 代替 A 可以允许惰性求值，你会在后面的章节中学到。当下，你可以认为这是一种直到真正需要时才创建一个 A 的办法。

现在，你的程序不需要 mock 就可以测试了。例如，以下是一个 buyDonuts 方

法的测试：

```
@Test
public void testBuyDonuts() {
    CreditCard creditCard = new CreditCard();
    Tuple<List<Donut>, Payment> purchase = DonutShop.buyDonuts(5, creditCard);
    assertEquals(Donut.price * 5, purchase._2.amount);
    assertEquals(creditCard, purchase._2.creditCard);
}
```

让你的程序成为函数式的另外一个好处是它更容易被复合。如果同一个人用你的初始代码购买多份，你只能一次次地调用银行（并付账）。通过新的函数式版本，你可以选择是每次购买立即支付还是为同一张信用卡合并支付。

为了把支付分组，你将需要函数式 `List` 类的附加方法（现在并不需要知道这些方法是怎么工作的，你将在第 5 章和第 8 章中详细地学习它们）：

```
public <B> Map<B, List<A>> groupBy(Function<A, B> f)
```

这个 `List` 类的实例方法接收一个从 `A` 到 `B` 的函数并返回一个 `map`，它由键值对组成，其中键的类型为 `B`，值的类型为 `List<A>`。换句话说，它通过信用卡把支付分组：

```
List<A> values()
```

这是 `map` 的实例方法，返回 `map` 里所有值的集合：

```
<B> List<B> map(Function<A, B> f)
```

这是 `List` 的实例方法，接收一个从 `A` 到 `B` 的函数并应用于 `A` 集合里的所有元素，从而得到一个 `B` 集合：

```
Tuple<List<A1>, List<A2>> unzip(Function<A, Tuple<A1, A2>> f)
```

这是 `List` 类的方法，接收一个从 `A` 到元组的函数。例如，可以是一个这样的函数：通过电子邮件地址，返回名称和域名组成的元组。在这个例子中的 `unzip` 方法，可以返回一个名称列表和域名列表组成的元组。

```
A reduce(Function<A, Function<A, A>> f)
```

这个 `List` 的方法用了一个把列表化简（`reduce`）为单值的操作。这个操作表示为 `Function<A, Function<A, A>> f`。这个记号看起来似乎有点怪异，但是你

将在第2章中学习它的含义。例如，它可以是一个加法。在这种情况下，它简单地表示为一个如 $f(a, b) = a + b$ 的函数。

通过这些方法，你现在可以创建一个通过信用卡把支付分组的全新方法，如清单 1.6 所示。

清单 1.6 通过信用卡把支付分组

```
package com.fpinjava.introduction.listing01_06;

import com.fpinjava.common.List;

public class Payment {

    public final CreditCard creditCard;
    public final int amount;

    public Payment(CreditCard creditCard, int amount) {
        this.creditCard = creditCard;
        this.amount = amount;
    }

    public Payment combine(Payment payment) {
        if (creditCard.equals(payment.creditCard)) {
            return new Payment(creditCard, amount + payment.amount);
        } else {
            throw new IllegalStateException("Cards don't match.");
        }
    }

    public static List<Payment> groupByCard(List<Payment> payments) {
        return payments
            .groupBy(x -> x.creditCard)
            .values()
            .map(x -> x.reduce(c1 -> c2 -> c1.combine(c2)));
    }
}
```

将一个 List<Payment> 转换为一个 Map<CreditCard, List<Payment>>, 在这个 map 中的每个 list 都会包含特定信用卡的所有支付

将每个 List<Payment> 化简为单值，生成一个最终的 List<Payment>

将 Map<CreditCard, List<Payment>> 转换为一个 List<List<Payment>>

请注意，你可以在 groupByCard 的最后一行使用方法引用，但是我使用了 lambda，因为它可能更易读。如果你喜欢方法引用，可以将那行代码替换为下面这行：

```
.map(x -> x.reduce(c1 -> c1::combine));
```


在清单 1.6 里, `c1 ->` 后面的部分是一个接收单参并将其传给 `c1.combine()` 的函数。而那其实就是 `c1::combine`——一个接收单参的函数。方法引用一般都会比 `lambda` 更加易读, 但并不绝对。

1.7 抽象到极致

如你所见, 函数式编程包含了通过复合没有副作用的纯函数来编写代码。这些函数可能表示为方法, 也可能是一等公民 (first-class) 的函数, 如上例中的 `groupBy`、`map` 和 `reduce` 方法的参数。一等公民的函数与方法不同, 可以由程序来操作。在大多数情况下, 它们被用作其他函数或方法的参数。你会在第 2 章中学到这是如何实现的。

然而这里最重要的概念是抽象。看看 `reduce` 方法。它接收一个操作作为参数, 并用其将列表简化为单值。这里的操作有两个相同类型的操作对象。除此之外, 它可以是任何操作。思考一个整型列表。你可以编写一个 `sum` 方法来对成员求和; 可以编写一个 `product` 方法来对成员求积; 还可以编写一个 `min` 或 `max` 方法来计算列表的最小值或最大值。而你可以对所有的这些计算重用 `reduce` 方法。这就是抽象。在 `reduce` 方法中, 你将所有操作的通用部分抽象出来, 并传递变量 (操作) 作为参数。

其实你还可以更进一步。相对生成不同于列表元素类型的更加通用的方法而言, `reduce` 方法是一个特例。例如, 可以通过一个字符列表创建出一个 `String`。为此你需要从一个给定值 (很可能是一个空字符串) 开始。在第 3 章和第 5 章中, 你将学会如何开发这个方法 (称为 `fold`)。还要注意的是, `reduce` 方法不能在一个空列表中正常工作。考虑一个整型数组——如果想要求和, 那就需要一个初始元素。如果列表是空的, 你应该返回什么? 当然, 你知道结果应该是 0, 但它只适用于求和, 并不适用于求积。

同样考虑一下 `groupByCard` 方法。它看起来像是一个只能用于通过信用卡把支付分组的业务方法。但是并非如此! 你可以用这个方法通过任意属性对任意集合里的元素分组, 所以这个方法应该被抽象出来并放到 `List` 类里以使其更容易被重用。

函数式编程的一个非常重要的部分就是将抽象推到极致。在本书的剩余部分, 你将会学到如何抽象许多东西, 以至于再也不需要定义它们了。例如, 你将学到如

何抽象循环，以至于你再也不需要编写循环了。你还会学到如何抽象并行化，以至于从顺序执行切换到并行执行仅仅是选择 `List` 类里的一个方法。

1.8 总结

- 函数式编程就是用函数来编程，返回结果，没有任何副作用。
- 函数式程序容易推断，也容易测试。
- 函数式编程提供了高层次的抽象和重用。
- 函数式程序比命令式程序更加健壮。
- 函数式程序由于避免了共享可变状态，因而在多线程环境中更加安全。

在Java中使用函数

本章要点

- 理解现实世界中的函数
- 在 Java 中表示函数
- 使用 lambda
- 使用高阶函数
- 使用柯里化函数
- 用函数式接口编程

为了理解函数式编程是如何工作的，我们会使用一些函数式库提供的函数式组件，还有 Java 8 标准库里可用的少许组件。不仅如此，我们还会关注如何构建它们，而不仅仅是如何使用。一旦掌握了这些概念，你便可以自行决定是使用自己的函数还是 Java 8 的标准函数，或者是现有的外部库之一。在本章你将会创建一个与 Java 8 的 `Function` 非常相似的 `Function`。为了让代码便于阅读，这个 `Function` 在处理类型参数上（避免通配符）会比较简单，但它会有一些 Java 8 缺失的强大能力。除了这些不同以外，它们可以互换。

你可能会在理解本章的某些代码时遇到麻烦。请不必担心，这是正常现象。出

于介绍函数的目的，我们需要引入一些诸如 List、Option 等函数式的结构。请耐心等待。所有未解释的组件都会在后续的章节中一一道来。

现在我要从现实世界和编程语言两方面入手，来详细解释什么是函数。函数并不仅是数学上或编程上的概念，函数是日常生活的一部分。我们经常会对所在的世界建模，这可不仅是编程。我们构建周围世界的表征，而这些表征通常都以随时改变自己状态的对象为基础。这种看待事物的方式是人类的天性。从状态 A 到 B 的迁移需要时间，并且需要耗费时间、精力或金钱。

还是以加法为例。大多数人将其视为耗时的计算（有时还得消耗脑力）。它有一个开始状态，一个转换过程（计算过程）和一个结果状态（加法的结果）。

为了对 345、765 和 34523 求和，我们当然需要进行计算。有些人可以在很短的时间内算出来，而有些人可能会耗时很久。有些人可能永远也计算不出来，或者算出一个错误的结果。有些人在他们的大脑中计算，有些人需要在纸上写下来。不管是用纸还是用脑，所有人可能都需要改变一些状态才能得到结果。但是对于 2 加 3，以上的这些我们全都不需要。大多数人都早已记住了答案，所以无须任何计算就可以立即给出结果。

这个例子展示了运算并不是必不可少的要素。它仅仅是计算一个函数的结果。但是在我们运算之前，结果就已经存在了。我们只是无法事先知道结果罢了。

函数式编程就是用函数来编程。为了能够做到这一点，首先我们需要知道函数是什么，不管是在现实世界中还是在我们选择的编程语言中。

2.1 什么是函数

虽然函数的概念在生活中无所不在，但是我们一般都把其当作一个数学上的对象。不幸的是，在日常生活中，我们经常混淆了函数和作用（effect）。更不幸的是，在许多编程语言中我们仍然会犯这个错误。

2.1.1 现实世界里的函数

在现实世界里，函数主要是数学上的概念。它是被称为函数定义域（function domain）的源集（source set）和被称为函数陪域（function codomain）的目标集（target set）之间的关系。定义域和陪域无须完全不同。例如，一个函数的定义域和陪域可以有相同的整数集。

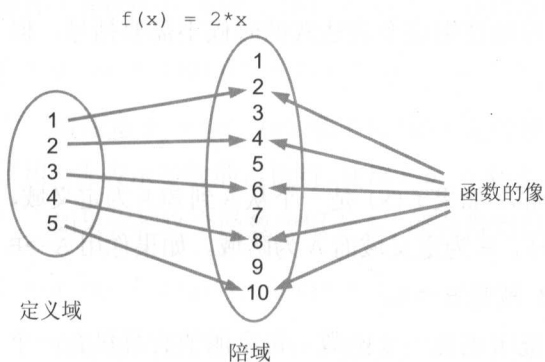
如何让两个集之间的关系成为函数

为了成为函数，关系需要满足一个条件：定义域内的所有元素都必须在陪域内有且仅有一个对应元素，如图 2.1 所示。

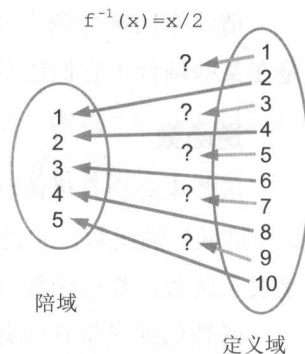
这个条件有一些有趣的含义：

- 定义域里不存在在陪域里没有对应值的元素。
- 定义域里的一个元素在陪域里不会有两个对应的元素。
- 陪域里的元素可能在源集里没有相对应的元素。
- 陪域里的元素可能在源集里有多多个相对应的元素。
- 在定义域里存在对应元素的陪域元素集被称为函数的像（image of the function）。

图 2.1 阐明了一个函数。



$f(x)$ 是一个从 N 到 N 的函数。



$f^{-1}(x)$ 在 N 的定义域上并不是一个函数。不过，在偶数集（ f 的像）的定义域上它就是一个函数。

图 2.1 函数定义域中的所有元素都必须有且仅有一个在陪域里相对应的元素。

例如，你可以定义函数

$$f(x) = x + 1$$

其中 x 是一个正整数。这个函数表示每个正整数和它的后继（successor）之间的关系。你可以给这个函数起任何名称。尤其是你可以赋予它一个能帮助记忆的名称，例如

$$\text{successor}(x) = x + 1$$

这似乎是一个好主意，但是你不应该无条件地信任一个函数名。你可以用另一个函数名取而代之，如下所示：

```
predecessor(x) = x + 1
```

这里不会发生任何错误，因为函数名称和定义之间是没有任何强制关系的。不过很明显，用这样的名称真是一个馊主意。

请注意，我们正在讨论函数是什么（它的定义）而不是函数能做什么。一个函数什么也不做。函数 `successor` 并不会对自己的参数加 1。你可以对整数加 1 来计算后继，但那样就不是函数了。这个函数

```
successor(x)
```

并不会对 x 加 1。它只是与 $x + 1$ 等价而已。简单来说，每当你碰到 `successor(x)` 表达式，就可以将其替换为 $(x + 1)$ 。

请注意用来分隔表达式的括号。在单独使用这个表达式的时候不需要括号，但是在某些场合中它们还是有用的。

逆函数

函数未必会有逆函数(inverse function)。如果 $f(x)$ 是一个从 A 到 B (A 为定义域， B 为陪域) 的函数，它的逆函数为 $f^{-1}(x)$ ， B 为定义域而 A 为陪域。如果你用 $A \rightarrow B$ 来表示函数，那逆函数（如果存在的话）就是 $B \rightarrow A$ 。

函数的逆函数在满足函数要求的情况下也是一个函数：每个源值有且仅有一个目标值。因此，对 `successor(x)` 取逆，你可以将其命名为 `predecessor(x)`（尽管你可以随便将它命名为 `xyz`）的关系，它在 N （包含 0 的正整数集）上并不是一个函数，因为 0 在 N 里并没有前驱（`predecessor`）。反过来，如果 `successor(x)` 被认为是带符号的整数集（正数和负数，标记为 Z ），那么 `successor` 的逆函数也是一个函数。

有些简单的函数没有逆函数。例如，这个函数

```
f(x) = (2 * x)
```

在 N 到 N 的定义上没有逆函数。如果你定义的是 N 到偶数集，那它就有逆函数了。

偏函数

没有在定义域中定义所有元素但是满足其他需求（定义域里不存在任何在陪域

里有多个元素与之相对应的元素)的关系一般称为偏函数 (partial function)。关系 $\text{predecessor}(x)$ 在 \mathbb{N} (包含 0 的正整数集) 上是一个偏函数, 但是在 \mathbb{N}^* (不包含 0 的正整数集) 上是一个全函数 (total function), 其陪域为 \mathbb{N} 。

偏函数在编程中相当重要, 因为许多 bug 都是由于将偏函数当作全函数使用而产生的。例如, $f(x) = 1/x$ 是一个从 \mathbb{N} 到 \mathbb{Q} (有理数) 的偏函数, 因为它对 0 没有定义。它是一个从 \mathbb{N}^* 到 \mathbb{Q} 的全函数, 也是一个从 \mathbb{N} 到 (\mathbb{Q} 与 error) 的全函数。通过在陪域里增加一个元素 (错误条件), 可以将偏函数转化为全函数。但是这样做的话, 这个函数需要有一种返回错误的方法。你能发现它与计算机编程的相似之处吗? 你将会看到把偏函数转换成全函数是函数式编程里的一个重要组成部分。

复合函数

函数就像积木, 可以复合为其他函数。函数 f 和 g 的复合函数标记为 $f \circ g$, 读作 f round g 。如果 $f(x) = x + 2$ 并且 $g(x) = x * 2$, 可得

$$f \circ g(x) = f(g(x)) = f(x * 2) = (x * 2) + 2$$

请注意 $f \circ g(x)$ 和 $f(g(x))$ 是等价的。但是写成复合函数 $f(g(x))$ 指出了用 x 来表示参数的占位符。使用 $f \circ g$ 来表示复合函数的话, 可以省略这个占位符。

如果你把这个函数应用于 5, 就会得到以下结果:

$$f \circ g(5) = f(g(5)) = f(5 * 2) = 10 + 2 = 12$$

有意思的是, $f \circ g$ 一般与 $g \circ f$ 不同, 虽然它们有时是等价的。例如:

$$g \circ f(5) = g(f(5)) = g(5 + 2) = 7 * 2 = 14$$

请注意, 应用函数的顺序与写函数的顺序正好相反。如果你写 $f \circ g$, 首先应用 g , 然后才是 f 。标准的 Java 8 函数定义了 `compose()` 和 `andThen()` 方法来表示这两种例子 (顺便提一句, 它们是冗余的, 因为 $f.\text{andThen}(g)$ 与 $g.\text{compose}(f)$ 或 $g \circ f$ 等价)。

多参函数

迄今为止, 我们只是讨论了单参函数。如果函数有多个参数会如何? 简单来说, 并没有多参函数这回事。还记得函数的定义吗? 一个函数是源集和目标集之间的关系。它并不是多个源集与一个目标集之间的关系。一个函数不允许有多个参数。

但是两个集的乘积本身是一个集，所以这样的函数可能确实有多个参数。让我们看看下面的函数：

$$f(x, y) = x + y$$

这似乎是一个 $N \times N$ 与 N 之间的关系，在这种情况下，它是一个函数。但是它只有一个参数，即 $N \times N$ 的元素。

$N \times N$ 是所有可能的整数对的集。这个集的元素就是一对整数，更通用的元组 (tuple) 表示多个元素的组合，而一对整数其实就是元组的一个特例。一对就是持有两个元素的元组。

元组用括号来表示，所以 $(3, 5)$ 是一个元组，也是一个 $N \times N$ 的元素。函数 f 可以应用于这个元组：

$$f((3, 5)) = 3 + 5 = 8$$

在这种情况下，你可以按照惯例删除一对括号：

$$f(3, 5) = 3 + 5 = 8$$

然而，它仍然是一个接收一个元组的函数，而不是两个参数的函数。

函数柯里化

元组函数可以用另一种方式来思考。可以认为函数 $f(3, 5)$ 是一个从 N 到 N 的函数集的函数。所以先前的例子可以这样重写

$$f(x)(y) = g(y)$$

其中

$$g(y) = x + y$$

在这种情况下，可以这样写

$$f(x) = g$$

它的意思是将函数 f 应用于参数 x ，结果是一个新的函数 g 。将函数 g 应用于 y 将会得到：

$$g(y) = x + y$$

当应用 g 的时候， x 就不再是一个参数了。它并不依赖于参数或者其他什么东西。

它就是一个常量。如果将其应用于 $(3, 5)$ ，你就会得到：

$$f(3)(5) = g(5) = 3 + 5 = 8$$

这里唯一的新知识就是 f 的陪域现在不是数字集了，而是函数集。将 f 应用于一个整数的结果是一个函数。将这个新函数应用于一个整数的结果是一个整数。

函数 $f(x)(y)$ 是 $f(x, y)$ 的柯里化形式。对一个元组函数（如果你喜欢可以称为多参函数）应用这种转换就称为柯里化（currying），源于数学家 Haskell Curry（虽然他并非是这种转换的发明者）。

偏应用函数

加法函数的柯里化形式可能看起来不那么自然，而且你可能会疑惑它是否对应着现实世界里的什么东西。毕竟，通过柯里化的版本，你需要单独考虑不同的参数。参数之一被视为第一个，将函数应用于它能得到一个新函数，这个新函数自己真的有用吗？还是说它仅仅是整体计算的一个步骤？

在加法的例子里，看起来没什么用。并且你其实可以从两个参数中的任意一个开始，并没有什么不同。虽然中间函数会不一样，但是最终结果总会是一样的。

现在思考一个接收一对值的新函数：

```
f(rate, price) = price / 100 * (100 + rate)
```

这个函数看起来跟以下函数等价：

```
g(price, rate) = price / 100 * (100 + rate)
```

现在让我们思考一下这两个函数的柯里化版本：

```
f(rate)(price)
g(price)(rate)
```

你知道 f 和 g 都是函数。但是 $f(\text{rate})$ 和 $g(\text{price})$ 都是什么呢？没错，它们的确是 f 应用于 rate 和 g 应用于 price 的结果。但是这些结果的类型都是什么？

$f(\text{rate})$ 是接收一个价格并返回一个价格的函数。如果 $\text{rate} = 9$ ，这个函数对一个价格应用 9% 的税，得到一个新价格。你可以将中间函数命名为 `apply9PercentTax(price)`，由于税率并不会经常变化，它可能还是一个挺有用的工具。

另一方面， $g(\text{price})$ 是一个接收税率并返回一个价格的函数。如果价格是

100 美元，它的新函数就是对 100 应用一个可变税率。你会如何命名这个函数呢？如果你想不出来一个有意义的名字，通常就意味着它没有什么用，虽然这取决于我们要解决的实际问题。

诸如 `f(rate)` 和 `g(price)` 这样的函数有时被称为偏应用函数，与 `f(rate, price)` 和 `g(price, rate)` 的形式有关。视参数的值而定，偏应用函数可能会有非常庞大的结果。我们会在后续章节再回到这个主题。

如果还不是很理解柯里化的概念，想象你正在外国旅行，用一个手持计算器（或者你的智能手机）来转换货币。当要计算价格的时候，你希望每次都需要输入汇率，还是把汇率存储在内存里？哪种办法更不容易出错？

没有作用的函数

请记住，纯函数只会返回一个值，不会做任何其他事情。它不会改变外界（外界是相对函数本身而言）的任何元素，不会改变自己的参数，也不会在出错时爆发（或者抛出异常等）。它可以返回一个异常或者其他的什么东西，例如一个错误消息，但必须将其返回，而不是将其抛出，不是写日志，也不是打印。

2.2 Java中的函数

在第 1 章中，你用了我称之为函数的东西，但实际上它们是方法。方法是一种在传统的 Java 里在某种程度上表示函数的方式。

2.2.1 函数式的方法

一个方法可以是函数式的，只要它满足纯函数的要求：

- 它不能修改函数外的任何东西。外部观测不到内部的任何变化。
- 它不能修改自己的参数。
- 它不能抛出错误或异常。
- 它必须返回一个值。
- 只要调用它的参数相同，结果也必须相同。

让我们来看一个例子。

清单 2.1 函数式的方法

```
public class FunctionalMethods {  
  
    public int percent1 = 5;  
    private int percent2 = 9;  
    public final int percent3 = 13;  
  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    public setPercent2(int value) {  
        percent2 = value;  
    }  
  
    public int mult(int a, Integer b) {  
        a = 5;  
        b = 2;  
        return a * b;  
    }  
  
    public int div(int a, int b) {  
        return a / b;  
    }  
  
    public int applyTax1(int a) {  
        return a / 100 * (100 + percent1);  
    }  
  
    public int applyTax2(int a) {  
        return a / 100 * (100 + percent2);  
    }  
  
    public int applyTax3(int a) {  
        return a / 100 * (100 + percent3);  
    }  
  
    public List<Integer> append(int i, List<Integer> list) {  
        list.add(i);  
        return list;  
    }  
}
```

你能说出哪些方法是纯函数吗？在阅读下面的答案之前先思考几分钟。想想方法内部的所有状态和过程。请记住，重要的是外界能观测到什么。别忘了考虑异常情况。

思考第一个方法：

```
public int add(int a, int b) {  
    return a + b;  
}
```

add 是一个函数，因为它返回的值总是取决于自己的参数。它并不修改自己的参数，也不以任何方式与外界交互。这个方法可能会在 $a+b$ 溢出最大的 `int` 值时出错，但它并不会抛出异常。结果是错误的（一个负数），但这是其他问题。每当函数被相同的参数调用，结果必须总是一致的。这并不意味着结果必须准确无误！

准确性 准确 (exact) 这个词本身并不表示什么东西。它通常只是表示结果与期望相符。所以说函数实现的结果是否准确，你需要知道实现的意图。通常除了函数名以外没有什么能够查明意图，而函数名可能是误解的起源。

思考第2个方法：

```
public int mult(int a, Integer b) {  
    a = 5;  
    b = 2;  
    return a * b;  
}
```

与 add 方法一样，mult 方法也是一个纯函数。你可能会觉得有些不可思议，因为它似乎改变了自己的参数。但是 Java 的方法参数是值传递的，意味着对它们重新赋值在方法外界观测不到。这个方法永远都会返回 10，因为它并不依赖于参数，所以没什么用处，但是这一点并没有违反要求。当这个方法被相同的参数多次调用时，它永远会返回相同的值。

顺便提一句，这个方法等价于一个无参方法。它是函数 $f(x) = 10$ 的一个特例，是一个常量。

现在思考 div：

```
public int div(int a, int b) {  
    return a / b;  
}
```

由于除数为 0 的时候会抛出异常，所以 div 方法并不是一个纯函数。为了让它成为函数，你可以检查第二个参数，当其为 0 时返回一个值。它必须是一个 `int` 类型，所以可能很难找到一个有意义的值，不过那是别的问题了。

思考第4个方法：

```
public int percent1 = 5;

public int applyTax1(int a) {
    return a / 100 * (100 + percent1);
}
```

applyTax1 方法似乎并不是一个纯函数,因为结果依赖于公共的 percent1 值,而这个值在两次调用间可能会不一致。因此,相同参数的两次调用可能会返回不同的值。percent1 可以理解为一个隐式的参数,但是这个参数并不像方法参数那样在同一时间决定。如果你在方法里只用一次 percent1 值,那就没什么问题,但是如果你读取了两次,它可能就会在两次读取操作之间发生变化。因此如果需要用这个值两次,你就必须读取一次并将其保存为局部变量。这就意味着 applyTax1 方法对于元组 (a, percent1) 来说是一个纯函数,但是对于 a 来说就不是了。

对比 applyTax2 方法:

```
private int percent2 = 9;

public int applyTax2(int a) {
    return a / 100 * (100 + percent2);
}
```

applyTax2 方法并没有什么不同,你可能会认为它是一个函数,因为 percent2 属性是私有的。但是它是可变的,通过 setPercent2 方法变化。因为 percent2 只被访问一次,所以 applyTax2 可以被认为是一个元组 (a, percent2) 的纯函数。但是它并不是 a 的纯函数。

现在思考第 6 个方法:

```
public final int percent3 = 13;

public int applyTax3(int a) {
    return a / 100 * (100 + percent3);
}
```

applyTax3 方法多少有些不同。在参数相同的情况下,这个方法总是会返回相同的值,因为它只依赖于自己的参数和 percent3 这个不可变的 final 属性。你可能会觉得 applyTax3 不是一个纯函数,因为结果并不只是依赖于方法的参数(纯函数的结果必须只依赖于自己的参数)。如果你认为 percent3 是一个额外参数,那这里就没有矛盾了。其实这个类本身也可以被认为是一个隐式的额外参数,因为在这个方法内部可以访问它的所有属性。

这是一个很重要的观念。所有的实例方法都可以通过在参数里增加外围类 (enclosing class) 的类型而变成一个静态方法。所以, applyTax3 方法可以被重写为

```
public static int applyTax3(FunctionalMethods x, int a) {  
    return a / 100 * 100 + x.percent3;  
}
```

这个方法可以从类内部被调用, 传一个 this 的引用作为参数, 如 applyTax3(this, a)。由于方法是公有的, 它也可以从外部被调用, 只需提供一个 FunctionalMethods 实例的引用。在这里, applyTax3 是一个元组 (this, a) 的纯函数。

终于到了我们的最后一个方法:

```
public List<Integer> append(int i, List<Integer> list) {  
    list.add(i);  
    return list;  
}
```

这个 append 方法在返回结果之前改变了参数, 而这个改变可在方法外界被观测到, 所以它不是一个纯函数。

对象标记与函数标记

你已经看到了, 实例方法访问类属性可以视为一个外围类实例的隐式参数。可以把不访问外围类实例的方法安全地标记为静态方法。访问外围类实例的那些方法也可以被标记为静态方法, 只需显式地标记它们的隐式参数 (外围类实例)。

思考一下第1章的 Payment 类:

```
public class Payment {  
  
    public final CreditCard cc;  
    public final int amount;  
  
    public Payment(CreditCard cc, int amount) {  
        this.cc = cc;  
        this.amount = amount;  
    }  
  
    public Payment combine(Payment other) {  
        if (cc.equals(other.cc)) {  
            return new Payment(cc, amount + other.amount);  
        } else {  
            throw new IllegalStateException(  
                "Can't combine payments to different cards");  
        }  
    }  
}
```



```

    }
}

```

combine 方法访问了外围类的 cc 和 amount 字段。结果就是它不能成为静态方法。这个方法将外围类视为一个隐式参数。

你可以将其变为显式参数，这样便可以使这个方法成为静态方法：

```

public class Payment {

    public final CreditCard cc;
    public final int amount;

    public Payment(CreditCard cc, int amount) {
        this.cc = cc;
        this.amount = amount;
    }

    public static Payment combine(Payment payment1, Payment payment2) {
        if (payment1.cc.equals(payment2.cc)) {
            return new Payment(payment1.cc, payment1.amount + payment2.amount);
        } else {
            throw new IllegalStateException(
                "Can't combine payments to different
cards");
        }
    }
}

```

你可以确定静态方法没有对外围作用域进行多余访问，但是它改变了方法的使用方式。

静态方法可以从类的内部被调用，只需传入 this 引用即可：

```
Payment newPayment = combine(this, otherPayment);
```

如果从类的外部调用方法，你需要使用类名：

```
Payment newPayment = Payment.combine(payment1, payment2);
```

这么做有少许不同，但是如果你需要复合方法调用，它们全都需要改变。如果你需要合并多个支付，以下是一个实例方法：

```

public Payment combine(Payment payment) {
    if (this.cc.equals(payment.cc)) {
        return new Payment(this.cc, this.amount + payment.amount);
    } else {
        throw new IllegalStateException(

```

```
        "Can't combine payments to different cards");  
    }  
}
```

可以使用对象标记：

```
Payment newPayment = p0.combine(p1).combine(p2).combine(p3);
```

它应该比这样要易读得多：

```
Payment newPayment = combine(combine(combine(p0, p1), p2), p3);
```

在前一个例子里合并更多支付也更简单。

2.2.2 Java 的函数式接口与匿名类

方法可以是函数式的，但是在函数式编程中，它缺少了一些可以用来表示函数的东西：除了应用于参数以外，它无法被控制。你无法将一个方法作为参数传递给另一个方法。结果就是你无法只是复合方法而不应用它们。你可以复合方法的应用，但无法复合方法本身。一个 Java 方法属于定义它的类，并且只能待在那里。

你可以通过从其他方法调用它们来复合方法，但是这应该在编写程序时就完成。如果你想在不同的情况下做不同的复合，就必须在写程序时规划好这些复合。你无法用程序在运行期间改变自身的方式来编写一段程序。难道可以吗？

是的，你可以！有时你可以在运行时注册处理器（handler）来处理特定的情况。你可以在处理器集合里增加处理器、移除或者改变使用顺序。怎么做到的呢？通过使用那些包含了你打算操作的方法的类。

在图形用户界面（GUI）上，你经常使用监听器（listener）来处理诸如移动鼠标、更改窗口大小或是输入文本等特定事件。这些监听器一般都是实现了一个特定接口的匿名类。你可以用相同的原理来创建函数。

假设你打算创建一个方法，返回整型参数的三倍。首先，你需要定义一个接口，里面有一个方法：

```
public interface Function {  
    int apply(int arg);  
}
```

然后你就可以实现这个方法来创建函数：

```
Function triple = new Function() {
```

```
@Override
public int apply(int arg) {
    return arg * 3;
}
};
```

接下来，这个函数可以应用于一个参数：

```
System.out.println(triple.apply(2));
6
```

我不得不承认这样做不太美观，旧的方法更加容易使用。如果你想创建另一个新方法，可以用相同的方式来处理：

```
Function square = new Function() {

    @Override
    public int apply(int arg) {
        return arg * arg;
    }
};
```

到目前还好，但是这样做又有什么优势呢？

2.2.3 复合函数

如果你把函数当成方法，复合它们似乎很简单：

```
System.out.println(square.apply(triple.apply(2)));
36
```

但它并不是复合函数，在本例中，你在复合函数的应用。复合函数是函数的二元操作，正如加法是数字的二元操作。因此你能够以编码的方式用一个方法来复合函数：

```
Function compose(final Function f1, final Function f2) {
    return new Function() {
        @Override
        public int apply(int arg) {
            return f1.apply(f2.apply(arg));
        }
    };
}
```

```
System.out.println(compose(triple, square).apply(3));
```

27

现在你开始看到这个概念是多么强大了吧！不过还有两个大问题。第一个问题是我们的函数只能接受整型（int）参数，并且返回整型。让我们先来搞定它吧。

2.2.4 多态函数

为了让我们的函数更加容易重用，你可以通过使用类型参数（parameterized types）来把它变成一个多态函数。Java 使用泛型来实现类型参数：

```
public interface Function<T, U> {  
    U apply(T arg);  
}
```

基于这个新接口，你可以重写为如下函数：

```
Function<Integer, Integer> triple = new Function<Integer, Integer>() {  
    @Override  
    public Integer apply(Integer arg) {  
        return arg * 3;  
    }  
};
```

```
Function<Integer, Integer> square = new Function<Integer, Integer>() {  
    @Override  
    public Integer apply(Integer arg) {  
        return arg * arg;  
    }  
};
```

如你所见，由于 Java 不能用 int 作为类型参数，我们把 int 类型改成了 Integer。希望自动装箱和拆箱机制可以透明地进行转换。

练习 2.1

用这两个新函数编写 compose 函数。

注意 每个练习后面都有答案，但是你应该尝试不看答案来完成练习。本书的网站上也包含了答案代码。这个练习比较简单，但是有一些会非常难，所以克制自己不看答案可能会有些困难。请记住，寻找答案越努力，你学到的东西就越多。

答案 2.1

```
static Function<Integer, Integer> compose(Function<Integer, Integer> f1,
                                           Function<Integer, Integer> f2) {
    return new Function<Integer, Integer>() {
        @Override
        public Integer apply(Integer arg) {
            return f1.apply(f2.apply(arg));
        }
    };
}
```

复合函数的问题

复合函数是一个非常强大的概念，但是如果用Java来实现，会有一个大隐患。复合几个函数没什么问题。但是思考一下，构建10 000个函数并把它们复合成一个。（可以通过折叠来完成，你将在第3章中学习这个操作。）

在命令式编程里，每个函数都在计算之后才把结果传递给下一个函数当作输入。但是在函数式编程里，复合函数意味着无须计算便直接构建结果函数。复合函数非常强大，因为函数无须计算就可以被复合。但是结果就是，在大量内嵌的方法调用中应用复合函数最终会导致栈溢出。可以用一个简单的例子来演示（使用下一节将会介绍到的lambda）。

```
int fnum = 10_000; Function<Integer, Integer> g = x -> x;
Function<Integer, Integer> f = x -> x + 1;
for (int i = 0; i < fnum; i++) {
    g = Function.compose(f, g);
};
System.out.println(g.apply(0));
```

当fnum为7500左右时，这段程序将会栈溢出。希望你不会经常复合几千个函数，不过你还是得知道这一点。

2.2.5 通过lambda简化代码

第二个问题就是，定义为匿名类的函数在用于编码时有些笨拙。如果你正在用Java 5到7，那真是太不幸了，因为没有其他办法。幸运的是，Java 8引入了lambda。

lambda 并不会改变定义 Function 接口的方式,但是它们让实现变得非常简单。

```
Function<Integer, Integer> triple = x -> x * 3;  
Function<Integer, Integer> square = x -> x * x;
```

lambda 并不只是简化语法,它还为代码编译提供了类型推断。lambda 和编写匿名类的传统方式的最大区别就是,它可以省略等式右边的类型。Java 8 关于类型推断的新功能使之成为可能。

在 Java 7 之前,类型推断只有在链式调用时才可用,就像这样:

```
System.out.println();
```

无须在此指定 out 的类型,Java 就可以找到它。如果不用链式写法,你就需要指定类型:

```
PrintStream out = System.out;  
out.println();
```

Java 7 通过 *diamond* 语法增加了一些类型推断:

```
List<String> list = new ArrayList<>();
```

你无须在此重复 ArrayList 的类型参数 String,因为 Java 能够通过查找定义来推断它的类型。lambda 也是一样:

```
Function<Integer, Integer> triple = x -> x * 3;
```

在本例中,Java 推断了 x 的类型。不过并不总能办到。当 Java 抱怨推断不出类型时,只能显式地写出来。于是括号便派上了用场:

```
Function<Integer, Integer> triple = (Integer x) -> x * 3;
```

指定函数类型

虽然 Java 8 中引入了 lambda 来轻松实现函数,但它还是缺乏一些简化编写函数类型的工具。从 Integer 到 Integer 的函数类型为

```
Function<Integer, Integer>
```

函数的实现写起来像下面这样:

```
x -> expression
```

如果能够应用类型简化一定是极好的，那便可以如下编写所有代码：

```
Integer -> Integer square = x -> x * x;
```

不幸的是，在 Java 8 里还办不到，并且这还不是你自己能够鼓捣出来的。

练习 2.2

使用 lambda 编写一个新版本的 compose 方法。

答案 2.2

直接用 lambda 替换掉匿名类就可以了。compose 方法的第一个版本的代码如下所示：

```
static Function<Integer, Integer> compose(Function<Integer, Integer> f1,
                                           Function<Integer, Integer> f2) {
    return new Function<Integer, Integer>() {
        @Override
        public Integer apply(Integer arg) {
            return f1.apply(f2.apply(arg));
        }
    };
}
```

所有你需要做的就是将 compose 方法的返回值替换为匿名类 apply 方法的参数，紧接着一个箭头 (->) 和 apply 方法的返回值：

```
static Function<Integer, Integer> compose(Function<Integer, Integer> f1,
                                           Function<Integer, Integer> f2) {
    return arg -> f1.apply(f2.apply(arg));
}
```

任意参数名皆可。图 2.2 展示了这个过程。

```
public static final Function<Integer, Integer> compose(final Function<Integer, Integer> f1,
                                                       final Function<Integer, Integer> f2) {
    return new Function<Integer, Integer>() {
        @Override
        public Integer apply(Integer arg) {
            return f1.apply(f2.apply(arg));
        }
    };
}

public static final Function<Integer, Integer> compose(final Function<Integer, Integer> f1,
                                                       final Function<Integer, Integer> f2) {
    return arg -> f1.apply(f2.apply(arg));
}
```

图 2.2 用 lambda 替换匿名类。

2.3 高级函数特性

你已经看过了如何创建 `apply` 和 `compose` 函数，也学过了函数可以表示为方法或对象。可是你还没有回答一个基础问题：为什么需要函数对象？不能只使用方法吗？在回答这个问题之前，你需要思考一下多参方法表示为函数式的问题。

2.3.1 多参函数怎么样

在 2.1.1 节中，我说过没有多参函数这回事，只有参数为元组的函数。元组的基数（cardinality）可以是任何你所需的东西。有几种元组拥有自己的特定名称：二元组（pair）、三元组（triplet）、四元组（quartet）等。它们还有其他名称，有些人喜欢称它们为 `tuple2`、`tuple3`、`tuple4` 等。不过我也说过，参数可以一个接一个地应用，除了最后一个参数以外，每个参数的应用都会返回一个全新的函数。

让我们来定义一个函数，用于对两个整数求和。你会把函数应用于第一个参数并返回一个函数。类型如下：

```
Function<Integer, Function<Integer, Integer>>
```

它似乎有点儿复杂，尤其是当你认为可以这么写的时候：

```
Integer -> Integer -> Integer
```

请注意，由于结合律，它等价于

```
Integer -> (Integer -> Integer)
```

左边的 `Integer` 是参数的类型，括号里的元素代表返回类型，此处显然是一个函数类型。如果把 `Function` 从 `Function<Integer, Function<Integer, Integer>>` 里删除，就会得到：

```
<Integer, <Integer, Integer>>
```

它们是完全一样的。Java 编写函数类型的方式确实有点累赘，但并不复杂。

练习 2.3

编写一个函数以对两个整数求和。

答案 2.3

这个函数将会接收一个 `Integer` 为参数并返回一个从 `Integer` 到 `Integer` 的函数，所以它的类型为 `Function<Integer, Function<Integer, Integer>>`。让我们把它命名为 `add`。它会用 `lambda` 来实现。最终的结果如下所示：

```
Function<Integer, Function<Integer, Integer>> add = x -> y -> x + y;
```

你会发现一行的长度很快就不够用了！`Java` 不支持类型别名，但是你可以通过继承来达到相同的效果。如果有许多相同类型的函数，你可以用一个更简短的标识来继承，就像下面这样：

```
public interface BinaryOperator extends
    Function<Integer, Function<Integer, Integer>> {}
BinaryOperator add = x -> y -> x + y;
BinaryOperator mult = x -> y -> x * y;
```

参数的数量没有限制，你可以定义接收任意数量参数的函数。正如我在本章的开始部分所说的，你在上面定义的 `add` 或 `mult` 函数，等价于接收元组为参数的函数的柯里化形式。

2.3.2 应用柯里化函数

你已经看到了如何编写柯里化函数的类型以及如何将其实现。但是如何应用它们呢？与任何函数一样。把函数应用于第一个参数，然后把结果应用于下一个参数，以此类推直至最后一个。例如，你可以将 `add` 函数应用于 3 和 5：

```
System.out.println(add.apply(3).apply(5));
8
```

你在此又错过了一些语法糖。如果可以只写函数名和参数就好了。`Scala` 就可以这么干：

```
add(3)(5)
```

Haskell 更好：

```
add 3 5
```

也许 `Java` 的未来版本也会支持这样的写法。

2.3.3 高阶函数

在练习 2.1 中，你为复合函数编写了一个方法。那个方法是函数式的，接收包含两个函数的元组为参数并返回一个函数。不过你可以用一个函数来代替这个方法！这个特殊版本的函数，接收函数为参数并返回函数，被称为高阶函数（higher-order function，即 HOF）。

练习 2.4

编写一个函数，用于复合在练习 2.2 里用过的 `square` 和 `triple` 这两个函数。

答案 2.4

如果你的步骤正确，这个练习还是比较容易的。要做的第一件事就是编写类型。这个函数会有两个参数，所以它将是一个柯里化函数。这两个参数和返回类型都是从 `Integer` 到 `Integer` 的函数：

```
Function<Integer, Integer>
```

你可以称其为 `T`。你要做的是创建一个接收类型 `T`（第一个参数）为参数的函数，并返回一个从 `T`（第二个参数）到 `T`（返回值）的函数。函数的类型如下所示：

```
Function<T, Function<T, T>>
```

如果把 `T` 替换为它的值，你就得到了真正的类型：

```
Function<Function<Integer, Integer>,  
        Function<Function<Integer, Integer>,  
                Function<Integer, Integer>>>
```

这里最主要的问题就是行的长度！现在让我们来补上实现，这可比类型要容易得多：

```
x -> y -> z -> x.apply(y.apply(z));
```

完整的代码如下所示：

```
Function<Function<Integer, Integer>,  
        Function<Function<Integer, Integer>,  
                Function<Integer, Integer>>> compose =  
    x -> y -> z -> x.apply(y.apply(z));
```

你可以在一行内写完代码！让我们用 `square` 和 `triple` 函数来测试这段代码：

```
Function<Integer, Integer> triple = x -> x * 3;  
Function<Integer, Integer> square = x -> x * x;  
Function<Integer, Integer> f = compose.apply(square).apply(triple);
```

在这段代码里，一开始你应用第一个参数，从而得到了一个新的函数并将其应用于第二个参数。结果还是一个函数，即复合两个函数参数的函数。将这个新的函数应用于（例如）2 所得到的结果，相当于首先将 `triple` 应用于 2 再将 `square` 应用于前面的结果（与函数复合的定义相关）：

```
System.out.println(f.apply(2));  
36
```

请注意参数的顺序：首先应用 `triple`，然后才将 `square` 应用于 `triple` 的返回结果。

2.3.4 多态高阶函数

虽然我们的 `compose` 函数还挺不错，可它只能复合从 `Integer` 到 `Integer` 的函数。如果你可以复合任意类型的函数，例如从 `String` 到 `Double`，或是从 `Boolean` 到 `Long`，那一定会更吸引人。不过那才是刚开始。一个完全多态的 `compose` 函数允许你复合 `Function<Integer, Function<Integer, Integer>>`，就像你在练习 2.3 中编写的 `add` 和 `mult` 那样。你还应该能够复合不同类型的函数，其中一个函数的返回类型与另一个函数的参数类型相同。

练习 2.5 (难)

编写一个 `compose` 函数的多态版本。

提示

你可能会在解答这道题时遇到两个问题。第一个是 `Java` 缺少对多态属性的支持。在 `Java` 中，你可以创建多态类、接口和方法，但是无法定义多态属性。解决办法就是将函数存储在方法、类或接口中，而非存储在属性中。

第二个问题是 `Java` 并不处理变体（`variance`），所以你可能会发现自己在尝试把 `Function<Integer, Integer>` 强制转换为 `Function<Object, Object>` 时发生编译错误。在这种情况下，只能显式地为 `Java` 指定类型。

变体

变体描述了类型参数和子类型之间的行为方式。协变 (covariance) 就是当 Red 是 Color 的子类时, Matcher<Red> 也被当作 Matcher<Color> 的子类。在这种情况下, Matcher<T> 就是对 T 的协变。如果反过来, Matcher<Color> 被当作 Matcher<Red> 的子类, 那么 Matcher<T> 就是对 T 的逆变 (contravariant)。在 Java 里, 虽然 Integer 是 Object 的子类, 但是 List<Integer> 并不是 List<Object> 的子类。你可能会觉得奇怪, 但是 List<Integer> 是一个 Object, 而不是一个 List<Object>。还有, Function<Integer, Integer> 并不是 Function<Object, Object>。(这个就没那么奇怪了吧!)

在 Java 里, 所有的类型参数的参数都是不可变 (invariant) 的。

答案 2.5

似乎第一步应该把练习 2.4 的例子“泛型化 (generify)”:

```
<T, U, V> Function<Function<T, U>,
    Function<Function<V, T>,
        Function<V, U>>> higherCompose =
    f -> g -> x -> f.apply(g.apply(x));
```

但它办不到, 因为 Java 不允许单独的泛型属性。为了成为泛型, 必须在定义类型参数的一个作用域里创建一个属性。只有类、接口和方法才能定义类型参数, 所以你只能在它们内部定义属性。最现实的就是一个静态方法:

```
static <T, U, V> Function<Function<T, U>,
    Function<Function<V, T>,
        Function<V, U>>> higherCompose() {
    return f -> g -> x -> f.apply(g.apply(x));
```

请注意, higherCompose() 方法没有参数, 并总是返回相同的值。它相当于是一个常量。它被定义成为一个方法事实上与这个观点无关。它并不是用于复合函数的方法, 它只是一个返回函数的方法, 返回的函数将会用于复合函数。

注意类型参数的顺序, 还有它们如何关联到 lambda 实现的参数上, 如图 2.3 所示。

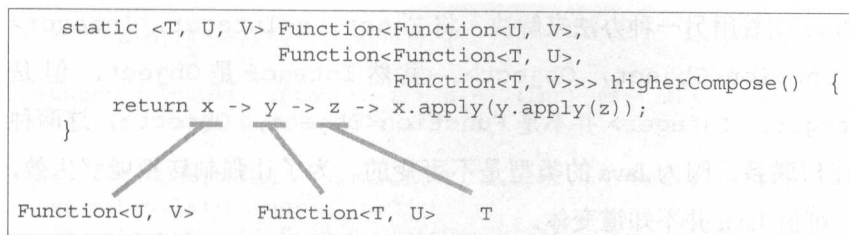


图 2.3 注意类型参数的顺序。

你可以赋予 `lambda` 参数一个更有意义的名字，例如 `uvFunction` 和 `tuFunction`，或者简单地叫作 `uv` 和 `tu`，但是你别这样做。名字是不可靠的。它们表达了（程序员）意图，仅此而已。你很容易会搞错名称而不自知：

```

static <T, U, V> Function<Function<U, V>,
                          Function<Function<T, U>,
                          Function<T, V>>> higherCompose() {
    return tuFunc -> uvFunc -> t -> tuFunc.apply(uvFunc.apply(t));
}

```

在本例中，`tuFunc` 是从 `U` 到 `V` 的函数，而 `uvFunc` 是从 `T` 到 `U` 的函数。

如果需要更多的类型信息，你可以在每个 `lambda` 参数前面简单地用括号把类型和参数包起来：

```

static <T, U, V> Function<Function<U, V>,
                          Function<Function<T, U>,
                          Function<T, V>>> higherCompose() {
    return (Function<U, V> f) -> (Function<T, U> g) -> (T x)
        -> f.apply(g.apply(x));
}

```

可能你现在想这样来使用此函数：

```
Integer x = Function.higherCompose().apply(square).apply(triple).apply(2);
```

可惜编译失败，生成如下错误：

```

Error:(39, 48) java: incompatible types: ...Function<java.lang.
Integer, java.lang.Integer> cannot be converted to ....Function<java.lang.
Object, java.lang.Object>

```

编译器提示它无法从 `T`、`U` 和 `V` 这些类型参数中推断出真正的类型，所以给这三个都用上了 `Object`。但是 `square` 和 `triple` 函数的类型都是 `Function<Integer, Integer>`。如果你认为通过这些信息足以推断出 `T`、`U` 和 `V` 的类型，那说明你比

Java 还聪明！Java 试着用另一种办法来解决，将 `Function<Integer, Integer>` 强制转换为 `Function<Object, Object>`。虽然 `Integer` 是 `Object`，但是 `Function<Integer, Integer>` 并不是 `Function<Object, Object>`，这两种类型之间没有任何联系，因为 Java 的类型是不可变的。为了让强制转换能够生效，类型需要协变，可惜 Java 并不知道变体。

解决办法是回到原来的问题上，告诉编译器 T、U 和 V 的真正类型是什么。可以通过在点和方法名之间插入类型信息来完成：

```
Integer x = Function.<Integer, Integer, Integer>higherCompose().apply(...
```

这多少有些不切实际，但是这并不是主要问题。更多时候，你会把一组像 `higherCompose` 这样的函数放到一个类库里去，并且通过静态引用（`static import`）来简化代码：

```
import static com.fpinjava. ... .Function.*;
...
Integer x = <Integer, Integer, Integer>higherCompose().apply(...;
```

不幸的是，编译失败！

练习 2.6（这回容易了！）

编写用于反向复合函数的 `higherAndThen` 函数，也就是说，`higherCompose(f, g)` 相当于 `higherAndThen(g, f)`。

答案 2.6

```
public static <T, U, V> Function<Function<T, U>, Function<Function<U, V>,
                                Function<T, V>>> higherAndThen() {
    return f -> g -> x -> g.apply(f.apply(x));
}
```

测试函数参数

如果你对参数的顺序有任何疑问，应该用不同类型的函数来测试这些高阶函数。测试从 `Integer` 到 `Integer` 的函数确实有一些模棱两可，因为从正反两边来复合函数都是可行的，所以不容易发现错误。这里有一个使用不同类型函数的测试：


```
public void TestHigherCompose() {  
    Function<Double, Integer> f = a -> (int) (a * 3);  
    Function<Long, Double> g = a -> a + 2.0;  
  
    assertEquals(Integer.valueOf(9), f.apply((g.apply(1L))));  
    assertEquals(Integer.valueOf(9),  
        Function.<Long, Double, Integer>higherCompose().apply(f).apply(g).  
            apply(1L));  
}
```

请注意, Java 无法推断类型, 所以你需要在调用 `higherCompose` 函数时提供类型。

2.3.5 使用匿名函数

迄今为止, 你一直在使用命名函数。这些函数由匿名类实现, 但是你创建的实例都已命名, 并有显式的类型。通常你并不会定义函数的名字, 而是将其当作匿名实例来使用。让我们来看一个例子。

无须如此编写:

```
Function<Double, Double> f = x -> Math.PI / 2 - x;  
Function<Double, Double> sin = Math::sin;  
Double cos = Function.compose(f, sin).apply(2.0);
```

你可以使用匿名函数:

```
Double cos = Function.compose(x -> Math.PI / 2 - x, Math::sin).apply(2.0);
```

你在此使用了 `Function` 类里定义的静态 `compose` 方法。不过这也适用于高阶函数:

```
Double cos = Function.<Double, Double, Double>higherCompose()  
    .apply(z -> Math.PI / 2 - z).apply(Math::sin).apply(2.0);
```

方法引用

除了 `lambda`, Java 8 中也引入了方法引用 (method reference), 即当 `lambda` 的实现是一个单参的方法调用时, 用于替换这个 `lambda` 的语法。例如:

```
Function<Double, Double> sin = Math::sin;
```

与此等价：

```
Function<Double, Double> sin = x -> Math.sin(x);
```

`sin` 在这里是一个 `Math` 类的静态方法。如果它是当前类的实例方法，你就可以这样写：

```
Function<Double, Double> sin = this::sin;
```

本书经常会使用这种代码，以生成一个方法的函数。

应该用匿名函数还是命名函数

除了那些用不了匿名函数的特殊场合外，你可以自行决定是用匿名函数还是命名函数。有一个基本规则：仅使用一次的函数可以被定义为匿名实例。但是只用一次，就意味着你只编写函数一次，并不意味着仅仅实例化一次。

在以下例子里，定义一个方法来计算 `Double` 值的余弦。这个方法使用两个匿名函数实现，因为你用了一个 `lambda` 表达式和一个方法引用：

```
Double cos(Double arg) {  
    return Function.compose(z -> Math.PI / 2 - z, Math::sin).apply(arg);  
}
```

不必担心创建匿名实例。`Java` 不会在每次调用函数时都创建新的对象。另外，实例化这样的对象代价很低。相反，你只应该从考虑代码的整洁性和可维护性的角度上决定是使用匿名函数还是命名函数。如果你考虑的是性能与重用性，那就应该尽量使用方法引用。

类型推断

匿名函数的类型推断也可能是一个问题。在上一个例子里，编译器可以推断出两个匿名函数的类型，是因为它知道 `compose` 方法接收两个函数为参数：

```
static <T, U, V> Function<V, U> compose(Function<T, U> f, Function<V, T> g)
```

但这样并不总能工作。如果你把第二个参数从方法引用替换为 `lambda`，

```
Double cos(Double arg) {  
    return Function.compose(z -> Math.PI / 2 - z,
```

```

    a -> Math.sin(a)).apply(arg);
}

```

迷茫的编译器就会显示以下错误消息：

```

Error:(64, 63) java: incompatible types: java.lang.Object cannot be
    converted to double
Error:(64, 44) java: bad operand types for binary operator '-'
    first type: double
    second type: java.lang.Object
Error:(64, 72) java: incompatible types: java.lang.Object cannot be
    converted to java.lang.Double

```

编译器是如此困惑，甚至还在第 44 列找到了一个根本不存在的错误！而实际上错误应该在第 63 列。虽然看起来很奇怪，但是 Java 无法猜出第二个参数的类型。为了让这段代码能够通过编译，你需要加上类型标记：

```

Double cos(Double arg) {
    return Function.compose(z -> Math.PI / 2 - z,
        (Function<Double, Double>) (a) -> Math.sin(a)).apply(arg);
}

```

这是使用方法引用的一个绝佳理由。

2.3.6 局部函数

你方才看到了可以在方法内部定义函数，但是不能在方法内部定义方法。

另一方面，函数可以用 `lambda` 的形式完美地定义在函数内部。嵌入 `lambda` 是最常见的例子，如下所示：

```

public <T> Result<T> ifElse(List<Boolean> conditions, List<T> ifTrue) {
    return conditions.zip(ifTrue)
        .flatMap(x -> x.first(y -> y._1))
        .map(x -> x._2);
}

```

如果你看不懂这段代码，无须担心。你将会在余下的章节中学到这类代码。请注意，`flatMap` 方法接收一个函数为参数（以 `lambda` 的形式），而这个函数（`->` 后面的代码）的实现定义了一个新的 `lambda`，与局部嵌入函数相对应。

局部函数并不总是匿名的。当作为辅助函数（`helper function`）使用时，它们一般都会被命名。在传统的 Java 中，使用辅助函数是一个常用的实践。这些方法可以让你通过抽象部分代码来简化它。相同的技术也可以用于函数，虽然隐式地使用匿

名 `lambda` 可能并不能让你注意到这一点。但是显式地声明局部函数总是可行的，正如以下示例，它基本上等价于上一个示例：

```
public <T> Result<T> ifElse_(List<Boolean> conditions, List<T> ifTrue) {
    Function<Tuple<Boolean, T>, Boolean> f1 = y -> y._1;
    Function<List<Tuple<Boolean, T>>, Result<Tuple<Boolean, T>>> f2 =
        x -> x.first(f1);
    Function<Tuple<Boolean, T>, T> f3 = x -> x._2;
    return conditions.zip(ifTrue)
        .flatMap(f2)
        .map(f3);
}
```

正如先前所说，这两种形式（即是否使用局部命名函数）有一点不同可能有时候会很重要。当涉及类型推断时，使用命名函数意味着需要显式地指定类型，这在编译器无法正确推断的时候是非常有必要的。

它不仅对编译器有用，对那些在类型方面不擅长的程序员来说也是一个莫大的帮助。显式地编写期望的类型有助于定位到与预期不符的地方。

2.3.7 闭包

你已经知道纯函数计算的返回值不应该依赖于参数以外的东西。Java 方法经常访问类成员，不仅会读取，甚至还会写入。方法也会访问其他类的静态成员。我已经说过，函数式方法是遵守引用透明性的方法，这意味着它们除了返回一个值以外不会有其他可观测到的作用。函数亦如是。函数只在没有可观测到的副作用时才是纯函数。

可是如果函数（还有方法）的返回结果不仅依赖于它们的参数，还依赖于外围作用域的元素呢？你已经看过了这样的例子，这些外围作用域里的元素可以被视为函数或方法的隐式参数。

`lambda` 还有附加要求：一个 `lambda` 访问的局部变量必须是 `final` 的。这并不是 `lambda` 的新要求，而是 Java 8 以前的版本对匿名类的要求，而 `lambda` 也需要遵守相同的约定，虽然它并没有那么严格。自 Java 8 起，从匿名类或是 `lambda` 访问的元素都是隐式 `final` 的；它们无须被声明为 `final` 来表明它们是不会被改变的。让我们来看看这个例子：

```
public void aMethod() {
```

```
double taxRate = 0.09;
Function<Double, Double> addTax = price -> price + price * taxRate;
...
}
```

在本例中,函数 addTax“封闭”了 taxRate 局部变量。只要变量 taxRate 不变,就能通过编译,没有必要显式地把变量声明为 final。

以下示例由于变量 taxRate 不再是隐式的 final 而不能通过编译:

```
public void aMethod() {
    double taxRate = 0.09;
    Function<Double, Double> addTax = price -> price + price * taxRate;
    ...
    taxRate = 0.13;
    ...
}
```

请注意,这个要求只适用于局部变量。以下代码可以顺利编译:

```
double taxRate = 0.09;

public void aMethod() {
    Function<Double, Double> addTax = price -> price + price * taxRate;
    taxRate = 0.13;
    ...
}
```

在这个例子里有一点很重要: addTax 并不是 price 的函数,因为它不会为相同的参数返回相同的结果。然而,它可以被视为一个元组 (price, taxRate) 的函数。

如果你把它们当作附加的隐式参数,那么闭包与纯函数可相互兼容。然而,它们可能会在重构代码,或是作为参数传递给其他函数时带来问题。这样会导致程序不易读并且不好维护。

用元组作为函数的参数是让程序更加模块化的一种方式:

```
double taxRate = 0.09;
Function<Tuple<Double, Double>, Double> addTax
    = tuple -> tuple._2 + tuple._2 * tuple._1;
System.out.println(addTax.apply(new Tuple<>(taxRate, 12.0)));
```

但是使用元组有些笨拙,因为 Java 并没有为此提供一个简单的语法,只是函数的参数里可以用括号而已。你只能为元组函数定义一个指定的接口,如下所示:

```
interface Function2<T, U, V> {  
    V apply(T t, U u);  
}
```

这个接口可以用于 lambda :

```
Function2<Double, Double, Double> addTax = (taxRate, price) -  
    > price + price * taxRate;  
double priceIncludingTax = addTax.apply(0.09, 12.0);
```

请注意, Java 中只有 lambda 才是允许你为元组使用 (x, y) 标记的唯一位置。可惜它无法用于其他场合, 例如从函数中返回一个元组。

你也可以用 Java 8 定义的 BiFunction 类, 它模拟了接收二元组参数的函数。还有 BinaryOperator, 相当于类型相同的二元组参数的函数, 还有 DoubleBinaryOperator, 接收的都是 double 原始类型的二元组。所有这些备选都挺好, 但如果需要三个或更多的参数, 那该怎么办呢? 可以定义 Function3、Function4, 如此这般。但是柯里化是一个更棒的解决方案。这就是为什么学习使用柯里化非常有必要, 而且正如你所见, 它非常简单:

```
double tax = 0.09;  
  
Function<Double, Function<Double, Double>> addTax  
    = taxRate -> price -> price + price * taxRate;  
  
System.out.println(addTax.apply(tax).apply(12.00));
```

2.3.8 部分函数应用和自动柯里化

上一个例子中的闭包和柯里化版本都可以得到相同的结果, 可以认为它们等价。可事实上, 它们在“语义上”有所不同。正如我所说, 这两个参数的作用截然不同。税率并不会经常变化, 而价格很可能在每次调用时会变化。在闭包版本中尤其如此。函数封闭了一个不会变化 (因为它是 final 的) 的参数。在柯里化版本中, 两个参数都可能会随每次调用而变化, 虽然税率并不会比闭包版本变化更频繁。

改变税率的需求还是挺常见的, 例如当产品类别或运输目的地不同时通常有几种税率。在传统的 Java 中, 这是通过把类转换成一个参数化的“税计算器”来实现的:

```
public class TaxComputer {  
  
    private final double rate;
```

```

public TaxComputer(double rate) {
    this.rate = rate;
}

public double compute(double price) {
    return price * rate + price;
}
}

```

这个类允许你为多种税率实例化多个 `TaxComputer`，并且能够随时重用这些实例：

```

TaxComputer tc9 = new TaxComputer(0.09);
double price = tc9.compute(12);

```

可以通过部分应用一个函数来达到同样的效果：

```

Function<Double, Double> tc9 = addTax.apply(0.09);
double price = tc9.apply(12.0);

```

这里的 `addTax` 是 2.3.7 小节最后的那个函数。

你可以看到柯里化和部分应用紧密地联系着。柯里化包括了把接收元组的函数替换为可以部分应用各个参数的函数。这是柯里化函数和元组函数最主要的区别。使用元组函数，所有的参数都在调用函数之前就计算出来了。使用柯里化版本，所有的参数都必须在完全应用函数之前确定，但是每个单独的参数都可以在函数部分应用它之前才计算。你不必将函数完全柯里化。一个接收三个参数的函数可以被柯里化为一个生成单参函数的二元组函数。

在函数式编程里，柯里化和部分应用函数用得如此频繁，抽象这些操作以允许自动使用便显得非常有帮助。在前面的章节中，只用了柯里化函数而非元组函数。这显示出了一大优势：部分应用函数绝对非常直观。

练习 2.7 (非常简单)

编写一个函数式方法来部分应用一个双参柯里化函数的第一个参数。

答案 2.7

你没什么可做的！方法的签名如下所示：

```

<A, B, C> Function<B, C> partialA(A a, Function<A, Function<B, C>> f)

```


你立即就能看到部分应用第一个参数与应用第二个参数（一个函数）都同样简单：

```
<A, B, C> Function<B, C> partialA(A a, Function<A, Function<B, C>> f) {
    return f.apply(a);
}
```

（如果你想看一个如何使用 `partialA` 的示例，请参考随书附带代码中本练习的单元测试。）

你可能注意到原来的函数是 `Function<A, Function<B, C>>` 类型，表示 $A \rightarrow B \rightarrow C$ 。如果打算部分应用第二个参数将会如何？

练习 2.8

编写一个方法来部分应用一个双参柯里化函数的第二个参数。

答案 2.8

有了上一个函数，这个问题的答案就是以下签名的方法：

```
<A, B, C> Function<A, C> partialB(B b, Function<A, Function<B, C>> f)
```

本练习稍微有点难度，但是如果你仔细思考了类型，那就还算简单。记住，你应该永远信任类型！不是所有的情况下它们都能立即给你一个方案，但是它们将把你引导至答案。这个函数只有一个可能的实现，所以如果你找到了一个可以成功编译的实现，你就可以确定它是正确的！

你知道你需要返回一个从 A 到 C 的函数。所以你可以开始这样编写实现：

```
<A, B, C> Function<A, C> partialB(B b, Function<A, Function<B, C>> f) {
    return a ->
```

这里的 `a` 是类型 A 的一个变量。在右箭头后面，你需要编写一个由函数 `f` 和变量 `a`、`b` 组成的表达式，并且它的值必须是从 A 到 C 的函数。函数 `f` 是一个从 A 到 $B \rightarrow C$ 的函数。所以你可以从应用 A 开始：

```
<A, B, C> Function<A, C> partialB(B b, Function<A, Function<B, C>> f) {
    return a -> f.apply(a)
```

这样你就得到了一个从 B 到 C 的函数。你需要一个 C ，而你已经拥有了一个 B ，所以再来一次，答案非常直观：

```
<A, B, C> Function<A, C> partialB(B b, Function<A, Function<B, C>> f) {
    return a -> f.apply(a).apply(b);
}
```

就是它了！其实你只要紧跟类型的步伐就可以了，几乎没有什么要做的。

正如我所说，最重要的事情是你有一个函数的柯里化版本。你很可能迅速学会如何直接编写一个柯里化函数。刚开始编写 Java 函数式程序时，一个常见的任务就是把多参方法转换为柯里化函数。这很简单。

练习 2.9 (非常简单)

将以下函数转换为一个柯里化函数：

```
<A, B, C, D> String func(A a, B b, C c, D d) {
    return String.format("%s, %s, %s, %s", a, b, c, d);
}
```

(我承认这个方法一点儿用也没有，它就是一个练习而已。)

答案 2.9

除了将逗号替换为右箭头以外，你还是没有什么好做的。尽管如此，谨记你需要把这个函数定义在接受类型参数的作用域里，还不能使用属性。你需要把它和所有需要的类型参数都定义在一个类、一个接口或一个方法里。

你会用方法来做。首先，编写方法的类型参数：

```
<A,B,C,D>
```

接下来，增加返回类型。刚开始似乎很难，但它只是难以阅读。写下 `Function<` 并在后面加上第一个参数类型和一个逗号：

```
<A,B,C,D> Function<A,
```

然后同样处理第二个参数类型：

```
<A,B,C,D> Function<A, Function<B,
```

继续直到没有其余参数了：

```
<A,B,C,D> Function<A, Function<B, Function<C, Function<D,
```

增加返回类型并且闭合所有尖括号：

```
<A,B,C,D> Function<A, Function<B, Function<C, Function<D, String>>>>
```

增加函数名和大括号：

```
<A,B,C,D> Function<A, Function<B, Function<C, Function<D, String>>>> f() {
}
```

对于实现来说，列出所需的尽量多的参数，用右箭头将它们分开（最后再加一个箭头）：

```
<A,B,C,D> Function<A, Function<B, Function<C, Function<D, String>>>> f() {
    return a -> b -> c -> d ->
}
```

最后，增加实现，它和原来的方法是一样的：

```
<A,B,C,D> Function<A, Function<B, Function<C, Function<D, String>>>> f() {
    return a -> b -> c -> d -> String.format("%s, %s, %s, %s", a, b, c, d);
}
```

柯里化一个元组函数也可以用同样的原则。

练习 2.10

编写一个从 `Tuple<A, B>` 到 `C` 的柯里化函数。

答案 2.10

你只要再次跟随类型就可以了。你知道方法会接收一个 `Function<Tuple<A, B>, C>` 类型为参数并返回 `Function<A, Function<B, C>>`，所以签名如下：

```
<A, B, C> Function<A, Function<B, C>> curry(Function<Tuple<A, B>, C> f)
```

现在，你需要在实现中返回一个双参的柯里化函数，所以可以这样开始：

```
<A, B, C> Function<A, Function<B, C>> curry(Function<Tuple<A, B>, C> f) {
    return a -> b ->
}
```

最终，你需要对返回类型求值。可以将函数 `f` 应用于一个用参数 `a` 和 `b` 构建的新 `Tuple`：

```
<A, B, C> Function<A, Function<B, C>> curry(Function<Tuple<A, B>, C> f) {
    return a -> b -> f.apply(new Tuple<>(a, b));
}
```

只要能够编译就还是不会出错。这种信心就是函数式编程的诸多优势之一！（并不总是这样，你将在下一章学习如何让它更经常发生。）

2.3.9 交换部分应用函数的参数

如果有一个双参函数，也许你会想要仅应用第一个参数来得到一个部分应用函数。比如说你有如下函数：

```
Function<Double, Function<Double, Double>> addTax = x -> y ->
    y + y / 100 * x;
```

可能你想首先应用税率来得到一个新的单参函数，之后便可以将其应用于任意价格：

```
Function<Double, Double> add9percentTax = addTax.apply(9.0);
```

接下来，当你想要为一个价格加税时，就可以这么做：

```
Double priceIncludingTax = add9percentTax.apply(price);
```

这样是不错，但如果初始函数是下面这样呢？

```
Function<Double, Function<Double, Double>> addTax = x -> y ->
    x + x / 100 * y;
```

在这个例子里，价格是第一个参数。仅应用于价格很可能不会有什么实际用途，但是如何才能仅应用于税率呢？（假设你访问不了实现。）

练习 2.11

编写一个方法来交换柯里化函数的参数。

答案 2.11

以下方法返回了一个参数反序的柯里化函数。它可以泛化为任何数量和任何顺序的参数：

```
public static <T, U, V> Function<U, Function<T, V>> reverseArgs(Function<T,
    Function<U, V>> f) {
    return u -> t -> f.apply(t).apply(u);
}
```

有了这个方法，你可以部分应用这两个参数中的任意一个。例如，有一个根据利率和金额计算贷款月供的函数：

```
Function<Double, Function<Double, Double>> payment = amount -> rate -> ...
```

很容易就可以创建一个根据固定金额和可变利率来计算月供的单参函数，或是

一个根据固定利率和可变金额来计算月供的函数。

2.3.10 递归函数

在诸多函数式编程语言中，递归函数是一项必备功能，虽然递归和函数式编程没有什么关系。有些函数式程序员甚至认为递归是函数式编程的 `goto` 功能，应该尽量避免使用。然而作为函数式程序员，必须掌握递归，哪怕你最终决定不去使用。

如你所知，Java 在递归上的能力有限。方法可以递归地调用自己，但是这也意味着每次递归调用时，计算的状态都被压入栈中，直至满足终止条件为止。此时所有先前的计算状态都被弹出栈，一个接一个地被赋值。栈的大小是可配置的，不过所有的线程都会使用相同的大小。默认的大小取决于 Java 的实现，从 32 位版本的 320KB 到 64 位实现的 1064KB，与存储对象的堆的大小相比，它们实在是微不足道。这样导致的结果就是递归的次数有限。

确定 Java 能处理多少次递归有些困难，因为它取决于入栈数据的大小，以及在递归处理开始时栈的状态。一般来说，Java 可以处理 5000 到 6000 次递归。

由于 Java 内部使用了记忆化 (memoization)，因此使得人为挑战这个极限成为可能。这个技术包括将函数或方法的返回结果存放在内存中以便加快未来的访问。如果先前存放过，Java 无须重新计算便可以直接从内存中获取结果。除了加快访问速度，这么做还能更快地找到终止状态，从而在一定程度上避免递归。我们将会在第 4 章再回到这个主题，你将在那里学到如何在 Java 里创建基于堆的递归。在本节的剩余部分，暂且认为 Java 的标准递归不会出问题。

定义一个递归方法很容易。`factorial(int n)` 方法可以被定义为：参数为 0 时返回 1，否则返回 $n * \text{factorial}(n - 1)$ 。

```
public int factorial(int n) {  
    return n == 0 ? 1 : n * factorial(n - 1);  
}
```

回想到当 n 在 5000 到 6000 的时候会导致栈溢出，所以请勿在生产环境中使用这类代码。

编写递归方法的确很容易。那么递归函数呢？

练习 2.12

编写一个递归的 `factorial` 函数。

提示

你不要试图编写一个匿名的递归函数，因为函数若要调用它自己，就必须有一个名字，而且必须在调用自己之前定义好那个名字。由于它调用自己时应该已经被定义好了，那就说明在你尝试定义它之前它就应该已经被定义好了！

答案 2.12

暂且放下这个先有鸡还是先有蛋的问题。把一个单参方法转换为函数是很直截了当的。类型为 `Function<Integer, Integer>`，实现应该与方法相同：

```
Function<Integer, Integer> factorial = n -> n <= 1 ? n : n * factorial.apply(n - 1);
```

现在是最棘手的部分。代码编译出错是因为编译器会抱怨 `Illegal self reference`（非法引用自己）。这是什么意思？简单来说，当编译器读到这段代码时，它正在定义 `factorial` 函数。在这个过程中，它遇到了对 `factorial` 函数的调用，可它却还未被定义好。

结果就是，不可能定义一个本地的递归函数。但是你能将这个函数声明为成员变量或静态变量吗？这并不能解决引用自己的问题，因为它等价于如下定义一个数字变量：

```
int x = x + 1;
```

这个问题可以通过先声明变量，再改变其值来解决。改变值可以在构造函数或者其他方法里实现，但是相比起来在初始化程序（`initializer`）里做要方便得多。如下所示：

```
int x;
{
    x = x + 1;
}
```

由于在初始化程序执行以前就已经定义好了成员变量，所以这段代码是可以工作的。一开始，变量会被初始化为默认值（`int` 为 0，函数为 `null`）。变量有时为 `null` 并不算是一个问题，因为初始化程序会在构造函数之前被执行，所以除非其他的初始化程序也用这个变量，否则就是安全的。这个小技巧可以用于定义函数：

```
public Function<Integer, Integer> factorial;
{
```

```
factorial = n -> n <= 1 ? n : n * factorial.apply(n - 1);
}
```

也可以用于定义静态函数：

```
public static Function<Integer, Integer> factorial;

static {
    factorial = n -> n <= 1 ? n : n * factorial.apply(n - 1);
}
```

这个小技巧的唯一问题就是字段无法被声明为 `final`，因为函数式程序员偏爱不可变性，所以有点儿讨厌。幸好还有另一个可用于这种场合的小技巧：

```
public final Function<Integer, Integer> factorial =
    n -> n <= 1 ? n : n * this.factorial.apply(n - 1);
```

通过在变量名前面增加 `this.`，就可以在 `final` 的同时引用自己。对于 `static` 的实现来说，只用把 `this` 替换为类名即可：

```
public static final Function<Integer, Integer> factorial =
    n -> n <= 1 ? n : n * FunctionExamples.factorial.apply(n - 1);
```

2.3.11 恒等函数

你已经见到了，在函数式编程里，函数是被当作数据来对待的。它们可以作为参数传递给其他函数，可以被函数返回，也可以用于操作，如同 `integer` 或 `double`。在后面的程序中，你还会把函数应用于操作，还需要一个中性元素或者说是恒等元素来满足这些操作。一个中性的元素就像是求和中的 0，求积中的 1，或者是字符串拼接中的空字符串。

可以把恒等函数命名为 `identity` 方法，添加到 `Function` 类的定义中，以返回恒等函数：

```
static <T> Function<T, T> identity() {
    return t -> t;
}
```

通过这个附加方法，我们的 `Function` 接口现在就完整了，如清单 2.2 所示。

清单 2.2 完整的 `Function` 接口

```
public interface Function<T, U> {
```

```
U apply(T arg);
```

```
default <V> Function<V, U> compose(Function<V, T> f) {
    return x -> apply(f.apply(x));
}
```

```
default <V> Function<T, V> andThen(Function<U, V> f) {
    return x -> f.apply(apply(x));
}
```

```
static <T> Function<T, T> identity() {
    return t -> t;
}
```

```
static <T, U, V> Function<V, U> compose(Function<T, U> f,
                                         Function<V, T> g) {
    return x -> f.apply(g.apply(x));
}
```

```
static <T, U, V> Function<T, V> andThen(Function<T, U> f,
                                         Function<U, V> g) {
    return x -> g.apply(f.apply(x));
}
```

```
static <T, U, V> Function<Function<T, U>,
                          Function<Function<U, V>,
                          Function<T, V>>> compose() {
    return x -> y -> y.compose(x);
}
```

```
static <T, U, V> Function<Function<T, U>,
                          Function<Function<V, T>,
                          Function<V, U>>> andThen() {
    return x -> y -> y.andThen(x);
}
```

```
static <T, U, V> Function<Function<T, U>,
                          Function<Function<U, V>,
                          Function<T, V>>> higherAndThen() {
    return x -> y -> z -> y.apply(x.apply(z));
}
```

```
static <T, U, V> Function<Function<U, V>,
                          Function<Function<T, U>,
                          Function<T, V>>> higherCompose() {
    return (Function<U, V> x) -> (Function<T, U> y) -> (T z) -> x.apply(y.apply(z));
}
```


2.4 Java 8的函数式接口

lambda 被用于接收特定接口的地方，Java 正是以此来决定调用哪个方法的。Java 并不对命名加以限制，有些语言则不然。唯一的限制是所用的接口必须要明确，这通常意味着它应该有且仅有一个抽象方法。（实际上会更复杂一些，因为有些方法不算在内。）这样的接口就是 SAM（single abstract method，单一抽象方法）类型，被称为函数式接口（functional interface）。

请注意 lambda 并不仅仅用于函数，在标准 Java 8 中可以使用许多函数式接口，虽然它们并不都与函数相关。以下是一些比较重要的接口。

- `java.util.function.Function` 与本章开发的 `Function` 很相似。为了让方法更有用，方法的参数中加了一个通配符。
- `java.util.function.Supplier` 等价于无参函数。在函数式编程里，它就是一个常量，所以一开始你可能会觉得它没那么有用，但是它有两个特定的用途：首先，如果它不是引用透明的（不是一个纯函数），便可以用于提供可变数据，例如时间或者随机数。（我们才不用这么不函数式的东西！）第二个用途更有意思，它允许惰性求值。我们会在后续的章节中频繁探讨这个主题。
- `java.util.function.Consumer` 并不是函数，而是作用。（这里并不是指副作用。使用 `Consumer` 的唯一结果就是作用，因为它什么东西也不返回。）
- `java.lang.Runnable` 也可以用于不接收任何参数的作用。一般最好为其创建一个指定的接口，因为 `Runnable` 应该用于线程，并且大多数语法检测工具都会在它被挪作他用时向你投诉。

Java 定义了许多其他的函数式接口（在 `java.util.function` 包里有 43 个），对于函数式编程而言大都没有什么用。其中有许多处理原始类型和其他的双参函数，还有用于操作（接收两个类型相同参数的函数）的特定版本。

我并不会在本书中讲解太多标准的 Java 8 函数，我是有意而为之的。本书并不是一本关于 Java 8 的书，而是一本关于函数式编程，正好以 Java 为例的书。你要学的是如何构造东西而非使用已经提供的组件。一旦掌握了这些概念，你便可以自行决定是使用自己的函数还是使用标准的 Java 8 函数。我们的 `Function` 与 Java 8 中的 `Function` 相似。为了简化本书所示的代码，它并没有为参数使用通配符。另一

方面, Java 8 的 Function 也没有定义 compose 和 andThen 这样的高阶函数, 它只有方法。除了这些不同以外, 这些 Function 实现都是可互换的。

2.5 调试lambda

lambda 的使用推动了一种写代码的新风格。曾经写成许多小短行的代码现在经常被替换为如下的一行代码:

```
public <T> T ifElse(List<Boolean> conditions, List<T> ifTrue, T ifFalse) {
    return conditions.zip(ifTrue).flatMap(x -> x.first(y -> y._1))
        .map(x -> x._2).getOrElse(ifFalse);
}
```

(ifElse 方法的实现在此由于页宽的限制而被拆成两行, 但是在代码编辑器中它其实在同一行中。)

在 Java 5 到 7 中, 这段代码只能用非 lambda 的方式编写, 如清单 2.3 所示。

清单 2.3 用一行 lambda 的方法转换为早期的 Java 版本

```
public <T> T ifElse(List<Boolean> conditions, List<T> ifTrue, T ifFalse) {
    Function<Tuple<Boolean, T>, Boolean> f1 =
        new Function<Tuple<Boolean, T>, Boolean>() {
            public Boolean apply(Tuple<Boolean, T> y) {
                return y._1;
            }
        };
    Function<List<Tuple<Boolean, T>>, Result<Tuple<Boolean, T>>> f2 =
        new Function<List<Tuple<Boolean, T>>, Result<Tuple<Boolean, T>>>() {
            public Result<Tuple<Boolean, T>> apply(List<Tuple<Boolean, T>> x) {
                return x.first(f1);
            }
        };
    Function<Tuple<Boolean, T>, T> f3 =
        new Function<Tuple<Boolean, T>, T>() {
            public T apply(Tuple<Boolean, T> x) {
                return x._2;
            }
        };
    Result<List<Tuple<Boolean, T>>> temp1 = conditions.zip(ifTrue);
    Result<Tuple<Boolean, T>> temp2 = temp1.flatMap(f2);
    Result<T> temp3 = temp2.map(f3);
    T result = temp3.getOrElse(ifFalse);
}
```

```
return result;  
}
```

显然，lambda 版本更加易读也容易修改。一般都认为 Java 8 之前的版本复杂得无法令人接受。但是当调试的时候，lambda 的版本问题更多。如果一行相当于原本的 20 行代码，那么如何才能有效地设置断点以找到潜在的错误呢？问题在于并非所有的调试器都强大到可以轻松应付 lambda。虽然问题总会随时间慢慢被解决，但同时你也需要找出其他的方案。一个简单的方案就是把单行的版本拆分为多行，如下所示：

```
public <T> T ifElse(List<Boolean> conditions, List<T> ifTrue, T ifFalse) {  
    return conditions.zip(ifTrue)  
        .flatMap(x -> x.first(y -> y._1))  
        .map(x -> x._2)  
        .getOrElse(ifFalse);  
}
```

这样你就可以为每一行设置断点。不仅非常实用，而且让代码更易读（还让书的排版更加容易）。但是它并不能解决我们的问题，因为每一行仍然有许多传统调试器不能完全理解的内容。

为了降低这个问题的严重程度，对每个组件进行全面的单元测试非常重要，这里的组件指的是每个方法，以及作为参数传递给方法的每个函数。这很容易。使用过的方法（按出现的顺序）有 List.zip、Option.flatMap、List.first、Option.map 和 Option.getOrElse。不管这些方法都是做什么的，它们都可以被全面测试。虽然现在你还不了解它们，但是在后续章节中构建 Option 和 List 组件时，还会编写 map、flatMap、first 和 zip 的实现，还有 getOrElse 方法（以及许多其他方法）。正如你即将看到的那样，这些方法都是纯函数式的。它们不能抛出任何异常并且总是返回预期的结果，仅此而已。所以，当完全测试过它们之后，就再也不会会有什么坏事发生了。

先前的例子使用了以下三个函数：

- `x -> x.first`
- `y -> y._1`
- `x -> x._2`

第一个无法抛出任何异常，因为 `x` 不能为 `null`（你将在第 5 章中了解到原因），

并且 `first` 方法也不会抛出异常。

第二个和第三个函数无法抛出 `NullPointerException`，因为你确定 `Tuple` 不能通过 `null` 参数构建。（回顾第 1 章中 `Tuple` 类的代码。）图 2.4 展示了这些函数的匿名形式。

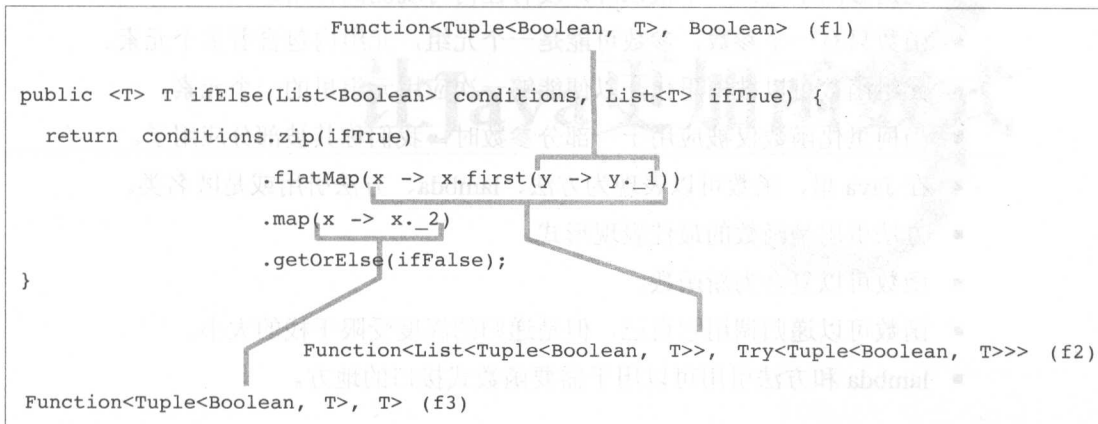


图 2.4 函数的匿名形式。

这正是函数式编程的亮点：如果组件都不会出错，那么整个程序也不会。在命令式编程中，组件可能在测试中运行良好，但在生产环境中由于一些不确定的因素而出错。如果组件的行为依赖于外部条件，那么你就无法完整地测试它。即使每个组件都没有问题，多个组件的组合仍然可能会导致程序表现欠佳。如果组件的行为确定，那么整个组合也将确定。

许多地方仍然可能出现错误。程序可能会由于组件复合错误而不按预期工作。不过实现错误并不会导致意外崩溃。如果程序崩溃了，那它永远都将如此，例如传递一个 `null` 给 `Tuple` 的构造函数。捕捉这种类型的错误无须使用调试器。

因此，调试大量使用 `lambda` 的函数式程序要比调试命令式程序难上几分。不过如果所有的组件都被验证过了，调试就变得不那么需要了。请记住，程序崩溃只能由抛出的异常导致。我们将会在第 6 章重新回到这里。现在你只要记着在默认情况下，一个抛出的异常或错误将只会使当前线程崩溃，而不是整个应用程序。甚至连 `OutOfMemoryError` 也不会使应用程序崩溃，所以作为程序员的你，需要处理好它。

2.6 总结

- 函数是源集和目标集之间的关系。它在源集的元素（定义域）和目标集的元素（陪域）之间建立联系。
- 纯函数除了返回一个值以外，没有任何可观测的作用。
- 函数只有一个参数，参数可能是一个元组，元组内包含着多个元素。
- 元组函数可以被柯里化，以便能够一次应用元组里的一个元素。
- 当柯里化函数仅被应用于一部分参数时，我们称其被部分应用了。
- 在 Java 里，函数可以表现为方法、lambda、方法引用或是匿名类。
- 方法引用是函数的最佳表现形式。
- 函数可以复合为新函数。
- 函数可以递归调用它自己，但是递归的深度受限于栈的大小。
- lambda 和方法引用可以用于需要函数式接口的地方。

让Java更加函数式

3

本章要点

- 让标准控制结构具有函数式风格
- 抽象控制结构
- 抽象迭代
- 使用正确的类型

你现在拥有了所需的全部类型的函数。正如你在第 2 章中所见到的，这些函数并不要求传统的 Java 编码规则为其破例。使用方法作为纯函数（又称函数式方法）与大多数所谓的 Java 最佳实践完全吻合。你还没有更改规则或是添加任何外来概念。你刚添加了一些关于函数式方法能做什么的限制：它们可以返回一个值，仅此而已。它们不能改变任何对象或外围作用域中的引用及参数。在本章的第 1 部分，你将学习如何将相同的原则应用于 Java 的控制结构上。

你还学习了如何创建表示函数的对象，以使这些函数可以作为参数传递给方法和其他函数。但是为了让这样的函数更加有用，你必须创建可以操作它们的方法或函数。在本章的第 2 部分，你将学习如何抽象集合操作和控制结构以使用函数的强大功能。

本章最后介绍的技术能够让你在处理业务问题时更加得心应手。

3.1 使标准控制结构具有函数式风格

控制结构是命令式编程的主要组成部分。命令式的 Java 程序员都不会相信无须使用 `if...else`、`switch...case`、`for`、`while` 和 `do` 循环即可编写程序。这些结构是命令式编程的本质。但在接下来的章节中，你将学习如何编写绝对没有控制结构的函数式程序。在本节中，你不会那么激进——我们只关注以更加函数式的风格来使用传统的控制结构。

你在第2章中学到的一点是，纯函数式的方法除了返回一个值以外不能做任何其他事情。它们不能改变外围作用域里的对象或引用。方法返回的值只能取决于它的参数，尽管方法可以读取外围作用域里的数据。在这种情况下，数据被认为是隐式参数。

在命令式编程中，控制结构定义了作用域，它们一般在作用域内工作，也就意味着它们具有作用。这个作用可能仅在控件结构的作用域内可见，也可能在外围作用域内可见。控制结构还可以访问外围作用域来取值。清单3.1展示了电子邮件地址验证的基本示例。

清单 3.1 简易电子邮件地址验证

```
final Pattern emailPattern =  
    Pattern.compile("^([a-z0-9._%+-]+@[a-z0-9.-]+\\.[a-z]{2,4})$");
```

```
void testMail(String email) {  
    if (emailPattern.matcher(email).matches()) {  
        sendVerificationMail(email);  
    } else {  
        logError("email" + email + "is invalid.");  
    }  
}
```

模拟发送电子邮件

```
void sendVerificationMail(String s) {  
    System.out.println("Verification mail sent to " + s);  
}
```

模拟记录消息

```
private static void logError(String s) {  
    System.err.println("Error message logged: " + s);  
}
```

if 条件“封闭”了 emailPattern 字段

如果条件满足，则发送电子邮件

如果条件不满足，则记录一条错误消息

在这个例子中，`if...else` 结构①在外围作用域里访问了 `emailPattern` 变量。

从 Java 语法的角度上看, 这个变量并没有必要被定义为 `final`, 但是如果打算让 `testMail` 方法具有函数式风格, 那它就有必要。

另一种办法是在方法内部声明邮件样式 (`emailPattern`), 但是这将导致每个方法调用都要编译它。如果样式可能在调用之间变化, 你应该使其成为方法的第二个参数。如果条件为 `true`, 则变量 `email` 就会应用一个作用②。这个作用包括发送验证电子邮件, 可能还要检查电子邮件地址是否不仅格式正确而且地址有效。在本例中, 这个作用就是通过将消息打印到标准输出 (`standard output`) 中来模拟④。如果条件为 `false`, 变量就会将其包含在错误信息里来应用一个不同的作用③。这个消息被记录起来⑤, 通过将消息打印到标准错误 (`standard error`) 中再次模拟。

3.2 抽象控制结构

清单 3.1 中的代码是纯命令式风格。你永远不会在函数式编程中找到这样的代码。虽然 `testMail` 方法由于不返回任何东西使其看上去像是一个纯作用, 但是它其实混合了数据处理与作用。你应该避免这种东西, 因为它会导致代码不可测试。让我们看看怎样来清理它。

可能你打算做的第一件事就是将计算与作用分开, 以便可以测试计算结果。这可以通过命令式来实现, 但我更喜欢用一个函数, 如清单 3.2 所示。

清单 3.2 使用一个函数来验证电子邮件

```
final Pattern emailPattern =
    Pattern.compile("^[a-z0-9._%+-]+@[a-z0-9.-]+\\.[a-z]{2,4}$");
final Function<String, Boolean> emailChecker = s ->
    emailPattern.matcher(s).matches();

void testMail(String email) {
    if (emailChecker.apply(email)) {
        sendVerificationMail(email);
    } else {
        logError("email " + email + " is invalid.");
    }
}
```

在外围作用域声明 `emailChecker` 函数

将 `emailChecker` 函数应用于待验证的字符串

现在可以测试程序的数据处理部分了 (验证 `email` 字符串), 因为你已经干净利落地将其从作用中分离出来。但是仍然还有许多问题。其一是你只处理了字符串是否合法的情况。但是如果接收的字符串为 `null`, 则会抛出 `NullPointerException` (NPE)。思考以下示例:


```
testMail("john.doe@acme.com");
testMail(null);
testMail("paul.smith@acme.com");
```

即使电子邮件地址合法，第三行也不会被执行，因为第二行抛出的 `NPE` 终止了线程。最好可以获取到一条日志消息，以指明发生了什么，并继续处理下一个地址。

如果接收到一个空字符串，则会出现另一个问题：

```
testMail("");
```

这不会导致错误，但地址将无法得到验证，并将记录以下消息：

```
email is invalid.
```

两个空格（“email”和“is”之间）指明了字符串为空。不过如下的一个特定消息更好：

```
email must not be empty.
```

为了处理这些问题，首先你要定义一个特殊的组件来处理计算结果，如清单 3.3 所示。

清单 3.3 管理计算结果的组件

```
public interface Result {
    // Result 接口表示计算的结果
}

public class Success implements Result {
    // Success 表示计算成功
}

public class Failure implements Result {
    // Failure 表示计算失败，它通过错误消息实例化
    private final String errorMessage;

    public Failure(String s) {
        this.errorMessage = s;
    }

    public String getMessage() {
        return errorMessage;
    }
}
```

现在你可以编写新版本的程序了，如清单 3.4 所示。

清单 3.4 错误处理得更好的程序

```
import java.util.regex.Pattern;

public class EmailValidation {

    static Pattern emailPattern =
        Pattern.compile("^[-_%+]+@[a-z0-9.-]+\.[a-z]{2,4}$");

    static Function<String, Result> emailChecker = s -> {
        if (s == null) {
            return new Result.Failure("email must not be null");
        } else if (s.length() == 0) {
            return new Result.Failure("email must not be empty");
        } else if (emailPattern.matcher(s).matches()) {
            return new Result.Success();
        } else {
            return new Result.Failure("email " + s + " is invalid.");
        }
    };

    public static void main(String... args) {
        validate("this.is@my.email");
        validate(null);
        validate("");
        validate("john.doe@acme.com");
    }

    private static void logError(String s) {
        System.err.println("Error message logged: " + s);
    }

    private static void sendVerificationMail(String s) {
        System.out.println("Mail sent to " + s);
    }

    static void validate(String s) {
        Result result = emailChecker.apply(s);
        if (result instanceof Result.Success) {
            sendVerificationMail(s);
        } else {
            logError(((Result.Failure) result).getMessage());
        }
    }
}
```

运行这段程序生成了预期的输出：

```
Error message logged: email this.is@my.email is invalid.
Mail sent to john.doe@acme.com
Error message logged: email must not be null
Error message logged: email must not be empty
```

但这仍然不令人满意。使用 `instanceof` 来确定结果成功与否实在难看。而使用强制转换来访问失败消息更甚于此。最糟糕的是，在 `validate` 方法中有一些程序逻辑无法测试。因为这个方法是一个作用，这意味着它并不返回一个值，而是改变了外界。

有解决这个问题的良方吗？有。你可以返回一个做同样事情的小程序，而不是发送电子邮件或是记录消息。不去执行

```
sendVerificationMail(s)
```

和

```
logError(((Result.Failure) result).getMessage());
```

而是返回一个指令，当这个指令被执行时，生成相同的结果。拜 `lambda` 所赐，你可以轻而易举地做到这一点。

首先，需要一个表示可执行代码的函数式接口：

```
public interface Executable {
    void exec();
}
```

你可以使用标准的 `Runnable` 接口，但是如果这个接口没有用于运行线程，大多数代码验证器都会发出警告。所以你要使用自己的接口。

你可以轻松地修改程序，如清单 3.5 所示。

清单 3.5 返回可执行函数

```
public class EmailValidation {

    static Pattern emailPattern =
        Pattern.compile("^[-_%+]+@[a-z0-9.-]+\\.[a-z]{2,4}$");

    static Function<String, Result> emailChecker = s ->
        s == null
            ? new Result.Failure("email must not be null")
            : s.length() == 0
                ? new Result.Failure("email must not be empty")
                : emailPattern.matcher(s).matches()
                    ? new Result.Success()
                    : new Result.Failure("email " + s + " is invalid.");

    public static void main(String... args) {
```

```

validate("this.is@my.email").exec();
validate(null).exec();
validate("").exec();
validate("john.doe@acme.com").exec();
}

private static void logError(String s) {
    System.err.println("Error message logged: " + s);
}

private static void sendVerificationMail(String s) {
    System.out.println("Mail sent to " + s);
}

static Executable validate(String s) {
    Result result = emailChecker.apply(s);
    return (result instanceof Result.Success)
        ? () -> sendVerificationMail(s)
        : () -> logError(((Result.Failure) result).getMessage());
}

```

① Executable 通过调用 exec() 来执行

② validate 方法现在返回一个结果，并且没有副作用了

validate 方法②现在返回 Executable 而不是 void 了。它不再有任何副作用，而是一个纯函数。当返回 Executable 时①，可以通过调用其 exec 方法来执行。

请注意，Executable 也可以通过传递给其他方法，或是存储起来以便稍后执行。特别是它还可以置于一个数据结构中并在所有的计算都完成之后顺序执行。这样你就可以将程序的函数式部分与改变环境的部分分离开来。

你还用三目运算符代替了 if...else 控制结构。这是一个喜好的问题。三目运算符是函数式的，因为它返回一个值并且没有副作用。相比之下，if...else 结构可以通过使它只改变局部变量来使其变成函数式，但是它仍然还会有副作用。如果你看到许多内嵌的 if...else 结构的命令式程序，问问自己是否可以很容易用三目运算符代替它们。通常这是衡量设计的函数式程度的一个优良指标。但是请注意，通过调用非函数式方法获取返回值也可以使三目运算符变成非函数式。

3.2.1 清理代码

你的 validate 方法现在是函数式的了，但它并不整洁。使用 instanceof 运算符经常是烂代码的代名词。另一个问题是可重用性较低。当 validate 方法返回一个值时，你没有执行与否以外的其他选择。如果打算重用验证部分并生成一个不

同的作用那该怎么办？

`validate` 方法不应该依赖于 `sendVerificationMail` 或 `logError`。它应该只返回一个结果，表示电子邮件是否有效，而你应该能够选择成功或失败所需的作用。也许你可能不希望应用该作用，而是想要复合其他的处理。

练习 3.1 (难)

尝试将验证逻辑从应用的作用中解耦。

提示

首先，你需要一个单一方法的接口，其方法表示一个作用。其次，由于 `emailChecker` 函数返回一个 `Result`，那么 `validate` 方法可以返回这个 `Result`。在这种情况下，你就不再需要 `validate` 方法了。第三，你需要将一个作用“绑定”到 `Result` 上。但是因为结果可能是成功或是失败，因此最好绑定两个作用并让 `Result` 类选择应用哪一个。

答案 3.1

首先要做的是创建一个表示作用的接口，如下所示：

```
public interface Effect<T> {  
    void apply(T t);  
}
```

也许你更喜欢用 Java 8 中的 `Consumer` 接口。虽然类名起得不太好，但是做的事情都一样。

然后你需要修改 `Result` 接口，如图 3.1 所示。

名字代表什么

许多伟大的作家都写过关于名字的作品。莎士比亚在《罗密欧与朱丽叶》中写道：¹

名字代表什么？我们所称玫瑰，

即使换个名字，依然芬芳如故。

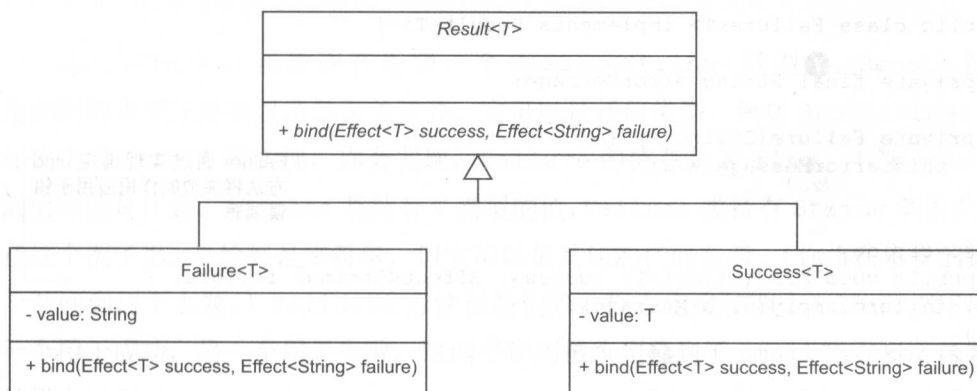
这美妙的两行里的内容让 Ferdinand de Saussure 和其他语言学家用了数百页来

¹ 威廉·莎士比亚，《罗密欧与朱丽叶》（1599），第2幕，场景2，http://shakespeare.mit.edu/romeo_juliet/romeo_juliet.2.2.html。

解释：名字和它所代表的内容之间的关系是随意的。结论就是程序员不应该相信名字。通常，选择名称是为了反映对象是什么或做什么。但即使对象只做一件事，它们仍然可能不匹配。

以 Java 接口为例。它们的命名应该表示对象是什么（Comparable、Cloneable、Serializable）或者它们做什么（Listener、Supplier、Consumer）。以此类推，Function 应该被重命名为 Applicable，并且有一个 apply 方法。一个 Supplier 应该定义一个 supply 方法，Consumer 应该消耗（consume）点什么并且有一个名为 consume 的方法。但是 Consumer 定义了一个 accept 方法而什么也不消耗，因为在接收（accept）了一个对象之后，这个对象仍然可用。

不要相信名称。务必相信类型。类型从不撒谎。类型是你的好朋友！



抽象类 **Result** 有两个实现，**Success** 和 **Failure**。注意，不管 **T** 是什么，**Failure** 所持有的值总是一个 **String**。在我们的示例里，**T** 是 **String**，但它也有可能是 **Email**。然而，**Failure<Email>** 的值总是一个持有相应错误消息的 **String**。

图 3.1 Result 接口的修改。

清单 3.6 展示了 **Result** 类修改后的版本。

清单 3.6 可以处理 **Effect** 的 **Result**

```

public interface Result<T> {

    void bind(Effect<T> success, Effect<String> failure);

    public static <T> Result<T> failure(String message) {
        return new Failure<>(message);
    }
}
  
```

Effect 由 bind 方法处理

```

public static <T> Result<T> success(T value) {
    return new Success<>(value);
}

public class Success<T> implements Result<T> {
    private final T value;

    private Success(T t) {
        value = t;
    }

    @Override
    public void bind(Effect<T> success, Effect<String> failure) {
        success.apply(value);
    }

    public class Failure<T> implements Result<T> {
        private final String errorMessage;

        private Failure(String s) {
            this.errorMessage = s;
        }

        @Override
        public void bind(Effect<T> success, Effect<String> failure) {
            failure.apply(errorMessage);
        }
    }
}

```

success 子类通过成功的值来初始化

success 通过实现绑定 bind 方法将成功的作用应用于值

Failure 通过实现绑定 bind 方法将失败的作用应用于错误消息

可以为 bind 方法选择任何你想要的名称。可以称它为 ifSuccess 或者 forEach。只有类型才重要。

现在你可以通过使用新的 Effect 和 Result 接口来清理程序,如清单 3.7 所示。

清单 3.7 一个更整洁的程序版本

```

public class EmailValidation {
    static Pattern emailPattern =
        Pattern.compile("^([a-z0-9._%+-]+@[a-z0-9.-]+\\.[a-z]{2,4}$");

    static Function<String, Result<String>> emailChecker = s -> {
        if (s == null) {
            return Result.failure("email must not be null");
        } else if (s.length() == 0) {
            return Result.failure("email must not be empty");
        } else if (emailPattern.matcher(s).matches()) {

```

① 函数 emailChecker 返回新的 Result 泛型


```

        return Result.success(s);
    } else {
        return Result.failure("email " + s + " is invalid.");
    }
};

public static void main(String... args) {
    emailChecker.apply("this.is@my.email").bind(success, failure);
    emailChecker.apply(null).bind(success, failure);
    emailChecker.apply("").bind(success, failure);
    emailChecker.apply("john.doe@acme.com").bind(success, failure);
}

static Effect<String> success = s ->
    System.out.println("Mail sent to " + s);
static Effect<String> failure = s ->
    System.err.println("Error message logged: " + s);
}

```

你不再需要
validate 方法了

3

2
函数 emailChecker
的返回值结果绑定
了两个 Effect

emailChecker 函数现在返回一个 Result<String> 泛型^①。Result 泛型是相同的类型还是错误消息无关紧要。它可以是任何类型，例如 Result<Email>。如果去看 Result 的实现，你会发现，Failure 的值总是 String，不管 success 的值可能是什么。success 类持有 T 类型的值，Failure 类持有 String 类型的值。在这个例子里，T 恰好是字符串，但它可以是其他的任何东西。（你将在本章的最后一节回到这个主题。）validate 方法已被删除，现在定义了两个 Effect 实例^③：一个用于成功，另一个用于失败。这两个作用被绑定^②到了 emailChecker 函数的结果上。

3.2.2 if ... else 的另一种方式

也许你想知道完全删除条件结构或运算符是否可行。你能编写一个没有这些结构的程序吗？看上去不太可能，因为许多程序员已经知道流程控制是编程的基本组成部分。但流程控制是一个命令式的编程概念。这个概念测试一个值并根据结果来决定下一步做什么。在函数式编程中，没有“下一步做什么”的问题，只有能够返回结果的函数。可以把最基本的 if 结构视为一个函数的实现：

```

if (x > 0) {
    return x;
} else {
    return -x;
}

```


这是一个 x 的函数。它返回 x 的绝对值。你可以如下编写这个函数：

```
Function<Integer, Integer> abs = x -> {
    if (x > 0) {
        return x;
    } else {
        return -x;
    }
}
```

与诸如以下的函数的区别

```
Function<Integer, Integer> square = x -> x * x;
```

是这个函数有两个实现，并且必须根据参数的值做选择。这个问题不大，但如果你的实现很多怎么办？最终会得到如清单 3.7 所示的那么多内嵌的 `if ...else` 结构，或是如清单 3.5 所示的那么多内嵌的三目运算符。还能做得更好吗？

练习 3.2

编写一个表示条件的 `Case` 类和相应的结果。条件由 `Supplier<Boolean>` 表示，其中 `Supplier` 是一个函数式接口，如下所示：

```
interface Supplier<T> {
    T get();
}
```

你可以使用 Java 8 或自己的 `Supplier` 实现。与条件相应的结果由 `Supplier<Result<T>>` 表示。为了持有它们，你可以用一个 `Tuple<Supplier<Boolean>, Supplier<Result<T>>>`。

`Case` 类应该定义三个方法：

```
public static <T> Case<T> mcase(Supplier<Boolean> condition,
                               Supplier<Result<T>> value)

public static <T> DefaultCase<T> mcase(Supplier<Result<T>> value)

public static <T> Result<T> match(DefaultCase<T> defaultCase,
                                Case<T>... matchers)
```

我用了 `mcase` 作为方法名，因为 `case` 是 Java 的关键字，`m` 表示匹配 (`match`)。当然，你可以选择任何名称。

第一个 `mcase` 方法定义了一种正常情况，接收一个条件和一个结果。第二个

mcase 方法定义了一种默认情况(default case), 由一个子类表示。第三个 match 方法, 选择一种情况。由于这个方法使用了一个变长参数 (vararg), 所以默认情况最先被传入, 但却最后才被使用!

另外, Case 类里应该用以下签名来定义私有的 DefaultCase 子类:

```
private static class DefaultCase<T> extends Case<T>
```

答案 3.2

我说过, 类必须要有一个表示条件的 Supplier<Boolean> 和一个表示结果的 Supplier<Result<T>>>。最简单的方式定义如下:

```
public class Case<T> extends Tuple<Supplier<Boolean>, Supplier<Result<T>>>{
    private Case(Supplier<Boolean> booleanSupplier,
                Supplier<Result<T>> resultSupplier) {
        super(booleanSupplier, resultSupplier);
    }
}
```

mcase 方法非常简单。第一个方法接收两个参数并创建一个新实例。第二个方法只接收第二个参数 (表示值的 Supplier), 并创建始终返回 true 的 Supplier 作为默认条件:

```
public static <T> Case<T> mcase(Supplier<Boolean> condition,
                                Supplier<Result<T>> value) {
    return new Case<>(condition, value);
}

public static <T> DefaultCase<T> mcase(Supplier<Result<T>> value) {
    return new DefaultCase<>(() -> true, value);
}
```

DefaultCase 类简单得不能再简单了。它只是一个标记类, 所以你只需创建一个调用 super 的构造函数即可:

```
private static class DefaultCase<T> extends Case<T> {
    private DefaultCase(Supplier<Boolean> booleanSupplier,
                        Supplier<Result<T>> resultSupplier) {
        super(booleanSupplier, resultSupplier);
    }
}
```

match 方法更加复杂, 不过有些言过其实, 因为它只有三行代码:

```
@SafeVarargs
```

```

public static <T> Result<T> match(DefaultCase<T> defaultCase,
                                Case<T>... matchers) {
    for (Case<T> aCase : matchers) {
        if (aCase._1.get()) return aCase._2.get();
    }
    return defaultCase._2.get();
}

```

正如我先前所说，由于第二个参数是变长参数，所以默认情况必须排在参数列表的首位，但是直到最后才被使用。通过逐个调用 `get` 方法进行计算来测试所有的情况。如果结果为 `true`，则在求值之后返回相应的值。如果都不匹配，则使用默认情况。

请注意，计算意味着对返回值求值。这次并不会应用任何作用。清单 3.8 展示了完整的类。

清单 3.8 用 Case 类匹配条件

```

public class Case<T> extends Tuple<Supplier<Boolean>, Supplier<Result<T>>>{

    private Case(Supplier<Boolean> booleanSupplier,
                 Supplier<Result<T>> resultSupplier) {
        super(booleanSupplier, resultSupplier);
    }

    public static <T> Case<T> mcase(Supplier<Boolean> condition,
                                   Supplier<Result<T>> value) {
        return new Case<>(condition, value);
    }

    public static <T> DefaultCase<T> mcase(Supplier<Result<T>> value) {
        return new DefaultCase<>(() -> true, value);
    }

    private static class DefaultCase<T> extends Case<T> {
        private DefaultCase(Supplier<Boolean> booleanSupplier,
                           Supplier<Result<T>> resultSupplier) {
            super(booleanSupplier, resultSupplier);
        }
    }

    @SafeVarargs
    public static <T> Result<T> match(DefaultCase<T> defaultCase,
                                     Case<T>... matchers) {
        for (Case<T> aCase : matchers) {
            if (aCase._1.get()) return aCase._2.get();
        }
        return defaultCase._2.get();
    }
}

```

`() -> true` 是一个表示 `Supplier` 的 `lambda`，它总是会返回 `true`。换句话说，它是一个“惰性”的 `true`。惰性对字面值（literal value）来说没有什么意义，但是你必须遵守 `DefaultCase` 构造函数的需求。

现在可以大大简化电子邮件验证应用程序的代码了。正如你在清单 3.9 中所见，它绝对没有包含控制结构。（请注意对 Case 和 Result 方法的静态导入。）

清单 3.9 没有控制结构的电子邮件验证应用程序

```
import java.util.regex.Pattern;
import static emailvalidation4.Case.*;
import static emailvalidation4.Result.*;

public class EmailValidation {

    static Pattern emailPattern =
        Pattern.compile("^[-a-z0-9._%+-]+@[a-z0-9.-]+\\.[a-z]{2,4}$");

    static Effect<String> success = s ->
        System.out.println("Mail sent to " + s);

    static Effect<String> failure = s ->
        System.err.println("Error message logged: " + s);

    public static void main(String... args) {
        emailChecker.apply("this.is@my.email").bind(success, failure);
        emailChecker.apply(null).bind(success, failure);
        emailChecker.apply("").bind(success, failure);
        emailChecker.apply("john.doe@acme.com").bind(success, failure);
    }

    static Function<String, Result<String>> emailChecker = s -> match(
        mcase() -> success(s),
        mcase() -> s == null, () -> failure("email must not be null"),
        mcase() -> s.length() == 0, () ->
            failure("email must not be empty"),
        mcase() -> !emailPattern.matcher(s).matches(), () ->
            failure("email " + s + " is invalid.")).
        );
}
```

默认
情况

可是等等。这是一个障眼法！你看不到任何控制结构是因为它们已经被隐藏在 Case 类中了，它包含了一个 if 指令甚至还有一个 for 循环。所以你是在作弊吗？并非如此。首先，你有一个整洁的循环和一个整洁的 if。不再有那些内嵌的 if 语句。其次，你已经抽象了这些结构。你现在可以编写许多条件应用程序，而无须写上一个 if 或者 for。但最重要的是，这才是你函数式编程之旅的起点。在第 5 章中，你将学习如何完全删除这两个结构。

在本章中，你将会看到如何对所有的控制结构进行通用的抽象。你已经完成了

对诸如内嵌的 `if...else` 语句这样的条件控制结构的抽象（与 `switch...case` 并没有什么不同）。下一步让我们看看如何同样处理循环。

3.3 抽象迭代

循环是迭代列表的结构。在 Java 中，循环也可以迭代集（set），甚至如索引循环般似乎没有迭代什么东西，不过迭代的总是列表。似乎在集上迭代两次并不会产生不同的结果，因为集在迭代时会应用一个顺序。甚至即使每次迭代时的顺序不同，也不会在一次迭代的过程中改变。所以从迭代的角度上可以把迭代的集当成是一个列表。

索引循环没有什么不一样——它通过计算索引来迭代一个列表。循环可以在计算所有参数之前退出，因为索引循环在索引上是惰性的。循环体总是惰性的，这意味着如果循环退出，剩余的元素将不会被处理。`if...else` 结构有着相似的行为。条件总会被计算，所以条件是及早（strict，与 lazy 相反）的，但在 `if` 和 `else` 分支中，只有一个会视条件而计算，所以 `if...else` 的分支也是惰性的。也许你认为 Java 是一门及早的语言，但并非如此。Java 的方法参数是及早的，但幸运的是它也有惰性时候。

回到循环，它们的主要作用是迭代列表中的全部元素，如下所示：

```
for(String email : emailList) {  
    // Do something with email;  
}
```

每当你想要处理一个列表时，使用这个结构或其他如 `while` 或是 `do...while` 的结构，并没有什么不同。它们只是迭代上的语法糖。甚至上文中的 `for` 循环也是以下代码的语法糖：

```
for (int i = 0; i < emailList.size(); i++) {  
    // do something with emailList.get(i)  
}
```

`while` 循环有所不同，因为只要条件验证通过它就会不停地迭代。它允许你在第一次迭代之前在某个条件上退出循环。`do...while` 循环也一样，但仅在第一次迭代之后。

重要的是应该在循环内做什么，那为什么要一遍又一遍地编写循环？为什么不

能仅说出想做什么并去做什么，而不是到处乱写控制结构、条件和索引呢？

举一个简单的例子。假设你有一个名字列表，打算返回以逗号分隔的字符串。你能够第一次就在纸上正确地编写程序吗？如果你是一个不错的程序员，我猜你可以。但是许多程序员需要编写代码，运行代码，修复常见的 bug，再次运行代码，修复边界情况下的 bug，然后再次运行程序直到正确为止。问题并不困难，但它是如此无趣以至于你经常不能第一次就搞定它。如果你总是第一次就能编写出正确的程序，恭喜你。你是一个好程序员，本节的其余部分可能不太适合你。但如果你是一个普通的程序员，请继续阅读。

你可能想在循环里做以下几件事情：

- 将每个元素转换为其他元素。
- 将元素聚合成单个结果。
- 根据元素自身的条件删除一些元素。
- 根据外部条件删除一些元素。
- 根据某些条件分组元素。

需要循环的各种操作都可以应用于集合，例如连接、压缩或解压缩。（压缩意味着从两个列表中获取元素并创建元组列表，解压缩是与之相反的操作）。

所有这些操作都可以被抽象。在第 5 章中，你将为实现所有这些抽象创建函数式的数据结构。现在，你就要开发一个这些抽象的库，以便将其应用于旧的 Java 集合上。

3.3.1 使用映射抽象列表操作

当应用于集合时，映射意味着对集合里的每个元素应用一个转换。这是在传统命令式编程里的常见做法：

```
List<Double> newList = new ArrayList<>();
for (Integer value : integerList) {
    newList.add(value * 1.2);
}
```

在这个例子里，一个增加 20% 的操作应用于 Integer 列表 (integerList) 里的每个元素。操作的结果是一个双精度类型，所以它被置于一个在循环开始以前创建的新列表中。虽然简单，但这段程序凸显了一些有意思的问题。

第一点是你可以将迭代从计算中分离出来。以下示例通过一个方法做到了这一点：

```
Double addTwentyPercent(Integer value) {  
    return value * 1.2;  
}  
  
List<Double> newList = new ArrayList<>();  
for (Integer value : integerList) {  
    newList.add(addTwentyPercent(value));  
}
```

你可以重用计算，但无法重用循环。为了允许重用，你可以把循环放到一个方法里，并传递一个函数来应用计算：

```
Function<Integer, Double> addTwentyPercent = x -> x * 1.2;  
  
List<Double> map(List<Integer> list, Function<Integer, Double> f) {  
    List<Double> newList = new ArrayList<>();  
    for (Integer value : list) {  
        newList.add(f.apply(value));  
    }  
    return newList;  
}
```

现在你可以用一个 Integer 列表和一个从 Integer 到 Double 的函数来调用 map 方法了，还能通过返回值获得一个新的 Double 列表。此外，你还可以自由重用该函数并通过不同的函数来调用 map 方法。

你可以通过使用泛型来极大地提高重用性：

```
<T, U> List<U> map(List<T> list, Function<T, U> f) {  
    List<U> newList = new ArrayList<>();  
    for (T value : list) {  
        newList.add(f.apply(value));  
    }  
    return newList;  
}
```

你可以将此方法包含在一个类库中，你将在这个类库中定义多个方法，从而允许抽象许多与列表相关的操作。可以称这个库为 CollectionUtilities。

3.3.2 创建列表

除了迭代，当程序员使用列表时需要一遍又一遍地重复其他的基本操作。最基本的操作是创建列表。Java 支持许多创建列表的方法，但是它们并不一致。

练习 3.3

编写方法来创建一个空列表、包含一个元素的列表、包含一个集合里所有元素的列表，以及一个从变长参数列表里创建列表的方法。所有的这些列表都是不可变的。

答案 3.3

这很直观，如以下代码所示：

```
public class CollectionUtilities {  
  
    public static <T> List<T> list() {  
        return Collections.emptyList();  
    }  
  
    public static <T> List<T> list(T t) {  
        return Collections.singletonList(t);  
    }  
  
    public static <T> List<T> list(List<T> ts) {  
        return Collections.unmodifiableList(new ArrayList<>(ts));  
    }  
  
    @SafeVarargs  
    public static <T> List<T> list(T... t) {  
        return Collections.unmodifiableList(Arrays.asList(Arrays.copyOf(t, t.length)));  
    }  
}
```

请注意，`list(List<T> ts)` 方法创建了参数列表的副本。这个防御性的副本是必要的，它用于确保该列表不会在以后被 `list` 方法的调用者所更改。还要注意，变长参数的版本可以使用数组为参数。在这种情况下，结果列表通过原始数组来备份。因此，改变数组的元素将会改变结果列表里的相应元素。这就是为什么你需要生成一个数组参数的副本。

还要注意的是，结果列表并非真的不可变。它们是不可变的视图里的可变列表，但是因为没人有权限访问这些可变列表，所以这样就足够了。它们只能在 `CollectionUtilities` 类中可变。

3.3.3 使用 head 和 tail 操作

对列表的函数式操作经常会访问列表的 `head`（或第一个元素）以及 `tail`（删除第一个元素后的列表）。

练习 3.4

创建两个方法分别返回列表的头部 (head) 和尾部 (tail)。不允许修改作为参数传递的列表。由于你需要创建列表的副本，所以还需要定义一个 copy 方法。tail 返回的列表必须是不可变的。

答案 3.4

head() 方法很简单。如果列表为空，则抛出异常。否则，读取索引 0 处的元素并将其返回。

copy 方法也很基本。它与创建列表的方法相同，以一个列表为参数。

tail 方法稍微复杂一些。它必须创建其参数的副本，删除第一个元素，并返回结果：

```
public static <T> T head(List<T> list) {
    if (list.size() == 0) {
        throw new IllegalStateException("head of empty list");
    }
    return list.get(0);
}

private static <T> List<T> copy(List<T> ts) {
    return new ArrayList<>(ts);
}

public static <T> List<T> tail(List<T> list) {
    if (list.size() == 0) {
        throw new IllegalStateException("tail of empty list");
    }
    List<T> workList = copy(list);
    workList.remove(0);
    return Collections.unmodifiableList(workList);
}
```

请注意，copy 是私有的，它返回一个可变的列表。为了从外界复制，你可以调用 list(List<T>)，以获得一个不可变的列表。还要注意的，当在空列表上调用 head 或 tail 时，本示例会抛出异常。这不是函数式风格，为了引用透明性，你应该总是捕获异常，而非抛出异常。然而，在目前这个阶段这样更简单。在第 5 章中，当查看函数式列表时，你会看到 head 和 tail 方法被声明为受保护的 (protected)。通过这种方式让它们只能在 List 类中使用，并且这个类不会泄漏出任何异常。

3.3.4 函数式地添加列表元素

在命令式程序中往 Java 的列表中添加一个元素是一个反复使用的基本操作：

```
list.add(element);
```

但是这个操作在函数式的程序中不可用，因为它会改变其参数，并且不返回修改后的列表。如果你因为它不会改变其参数而认为它是函数式的，请回忆一下你在第 2 章中所学的：这是对象标记。列表本身是 add 方法的隐式参数，因此它等价于：

```
add(list, element);
```

将这个方法转换为函数式的风格很简单。你可以称其为 append：

```
public static <T> List<T> append(List<T> list, T t) {  
    List<T> ts = copy(list);  
    ts.add(t);  
    return Collections.unmodifiableList(ts);  
}
```

append 方法创建其第一个参数的防御性副本（通过调用先前已经定义好的 copy 方法），向其添加第二个参数，然后返回包装在不可变视图中的已修改列表。你很快就会有机会在无法使用 add 的地方使用这个 append 方法。

3.3.5 化简和折叠列表

列表折叠（fold）通过使用一个特定操作来将列表转换为单值。结果可以是任何类型——不必与列表的元素类型相同。折叠的结果类型若是与列表元素相同，则是一种称为化简（reduce）的特殊情况。对整型列表的元素进行求和是化简的一种简单情况。

你可以在从左到右或从右到左这两个方向上折叠列表，视所用的操作而定：

- 如果操作可交换，则两种折叠方式是等价的。
- 如果操作不可交换，则两种折叠方式会给出不同的结果。

折叠操作需要一个起始值，即操作的中性元素或单位元（identity element），该元素作为累加器（accumulator）的起始值。当计算完成时，累加器内即包含结果。另一方面，只要列表不为空，也可以在没有起始元素的情况下执行化简，因为第一个（或最后一个）元素将作为起始元素。

用加法化简数字列表

假设你有一个列表 (1, 2, 3, 4), 并打算对所有元素求和。第一种方法是将累加器置于运算对象的左侧:

```
((0 + 1) + 2) + 3 + 4 = 10
```

也可以从另一侧进行处理:

```
1 + (2 + (3 + (4 + 0))) = 10
```

结果完全相同。对于乘法来说也是如此, 但你必须使用单位元 1 作为累加器的初始值。

将列表折叠成字符串

现在让我们通过应用于字符列表 ('a', 'b', 'c') 的不同操作来做同一件事。这里使用的操作如下:

```
"x" + 'y' = "xy"
```

首先, 让我们从左侧折叠:

```
(("" + 'a') + 'b') + 'c' = "abc"
```

现在让我们从右侧试试相同的做法:

```
'a' + ('b' + ('c' + "")) = "abc"
```

从右侧折叠行不通, 因为左操作数是一个字符, 而右侧的是一个字符串。因此, 你必须进行如下更改操作:

```
'x' + "y" = "xy"
```

在这种情况下, 字符被前置 (prepend) 到字符串上, 而非后置 (append)。第一个折叠称为左折叠 (left fold), 意味着累加器在操作的左侧。当累加器在右侧时, 它是一个右折叠 (right fold)。

理解左右折叠之间的关系

你可能会认为右折叠可以用左折叠来定义。让我们用一个称为反递归 (corecursion) 的不同形式来重写右折叠操作:

```
((0 + 3) + 2) + 1 = 6
```

对于递归和反递归而言，每一步的计算都取决于上一步。但是递归从最后一步开始，并定义它与上一步的关系。为了能够终止，它还必须定义基本步骤。另一方面，反递归从第一步开始，并定义它与下一步的关系。并不需要基本步骤，因为它就是第一步。

从这一点上来看，似乎右折叠列表等价于将元素反序之后再左折叠列表。

可是请稍等。求和是一个满足交换律的操作。如果使用不可交换的操作，你还需要更改操作。如果不这样做，可能会根据类型而遇到两种不同的情况。如果操作的运算对象类型不同，则无法编译。另一方面，如果操作的运算对象类型相同，但不是可交换的，则计算虽成功但会得到错误的结果。因此 `foldLeft` 和 `foldRight` 具有以下关系，其中 `operation1` 和 `operation2` 在运算对象相同、顺序相反时得到相同的结果：

```
foldLeft(list, acc, x -> y -> operation1)
```

等价于

```
foldRight(reverse(list), acc, y -> x -> operation2)
```

如果操作是可交换的，`operation1` 和 `operation2` 就是相同的。否则，如果 `operation1` 是 `x -> y -> compute(x, y)`，则 `operation2` 就是 `x -> y -> compute(y, x)`。

想想用于反转列表的 `reverse` 函数。你知道它如何用 `foldLeft` 来表示吗？这正是函数式编程之美的一部分。随处都可以发现抽象。现在让我们来看看如何将它应用于传统的 Java 列表上。

练习 3.5

创建一个用于折叠整型列表的方法，例如对列表的元素求和。该方法将接收一个整型列表、一个整型初始值和一个函数为参数。

答案 3.5

初始值取决于应用的操作。该值必须是中性的，或者说是操作的单位元。该操作表示为一个柯里化函数，如你在第 2 章中所学：

```
public static Integer fold(List<Integer> is, Integer identity,
                          Function<Integer, Function<Integer, Integer>> f) {
```

```

int result = identity;
for (Integer i : is) {
    result = f.apply(result).apply(i);
}
return result;
}

```

静态导入 `CollectionUtilities.*` 后，可以如下调用这个方法：

```

List<Integer> list = list(1, 2, 3, 4, 5);
int result = fold(list, 0, x -> y -> x + y);

```

这里的 `result` 等于 15，即 1、2、3、4 和 5 的和。用乘号替换加号并用 1（乘法的单位元）替换 0，即可得到 $1 \times 2 \times 3 \times 4 \times 5 = 120$ 的结果。

左折叠示例

你刚刚定义的操作被命名为 `fold`，因为左右折叠对整型求和或求积都能得到相同的结果。但是如果你打算使用其他函数，或者打算让折叠方法变得通用，那就必须区分左右折叠。

练习 3.6

将 `fold` 方法归纳为 `foldLeft`，以便它可以应用左折叠于任意类型的元素列表。要测试方法是否正确，将其应用于以下参数，

```

List<Integer> list = list(1, 2, 3, 4, 5);
String identity = "0";
Function<String, Function<Integer, String>> f = x -> y -> addSI(x, y);

```

其中，`addSI` 方法定义如下：

```

String addSI(String s, Integer i) {
    return "(" + s + " + " + i + ")";
}

```

确认你得到了以下输出：

```
(((((0 + 1) + 2) + 3) + 4) + 5)
```

请注意，`addSI` 方法允许你验证参数的顺序是否正确。直接使用 `"(" + s + " + " + i + ")"` 表达式验证不了，因为反转参数将只会改变 `+` 号的含义，而不会改变结果。

答案 3.6

命令式的实现相当简单：

```
public static <T, U> U foldLeft(List<T> ts, U identity,
                                Function<U, Function<T, U>> f) {
    U result = identity;
    for (T t : ts) {
        result = f.apply(result).apply(t);
    }
    return result;
}
```

泛型的版本也可以用于整型操作，所以专门的整型版本用处不大。

右折叠示例

如前所述，左折叠是一个反递归操作，因此很容易通过一个命令式的循环来实现。另一方面，右折叠是一个递归操作。要测试你的临时实现，可以用你在左折叠中使用的办法。你将用以下参数来测试实现，

```
List<Integer> list = list(1, 2, 3, 4, 5);
String identity = "0";
Function<Integer, Function<String, String>> f = x -> y -> addIS(x, y);
```

其中 addIS 方法被定义为：

```
private static String addIS(Integer i, String s) {
    return "(" + i + " + " + s + ")";
}
```

验证输出如下：

```
(1 + (2 + (3 + (4 + (5 + 0)))))
```

练习 3.7

编写一个 foldRight 方法的命令式版本。

答案 3.7

右折叠是一个递归操作。为了用命令式的循环来实现它，你需要反序处理列表：

```
public static <T, U> U foldRight(List<T> ts, U identity,
                                Function<T, Function<U, U>> f) {
    U result = identity;
    for (int i = ts.size(); i > 0; i--) {
```

```

        result = f.apply(ts.get(i - 1)).apply(result);
    }
    return result;
}

```

练习 3.8

编写 `foldRight` 的递归版本。注意，一个初始递归的版本并不能在 Java 中工作得天衣无缝，因为它使用栈来累积中间计算。在第 4 章中，你将学习如何创建栈安全的递归（`stack-safe recursion`）。

提示

你应该将该函数应用于列表的 `head` 和折叠 `tail` 的结果。

答案 3.8

初始版本至少能处理 5000 个元素，对于练习来说已经足矣：

```

public static <T, U> U foldRight(List<T> ts, U identity,
                                Function<T, Function<U, U>> f) {
    return ts.isEmpty()
        ? identity
        : f.apply(head(ts)).apply(foldRight(tail(ts), identity, f));
}

```

基于堆的递归 答案 3.8 并不是尾递归，因此它不适用于以堆替栈的优化。我们将在第 5 章讨论基于堆的实现。

反转列表

反转列表有时挺实用的，但是这个操作的性能不太好。最好能找到其他不需要反转列表的办法，但并不总能找到。

用命令式的实现来定义 `reverse` 方法很容易通过反向遍历列表实现。但是，你要小心别把索引搞乱：

```

public static <T> List<T> reverse(List<T> list) {
    List<T> result = new ArrayList<T>();
    for(int i = list.size() - 1; i >= 0; i--) {
        result.add(list.get(i));
    }
    return Collections.unmodifiableList(result);
}

```

有许多方式都可行。例如，你可以从 `list.size()` 开始迭代，并使用 `i > 0`

作为条件。需要在之后用 $i-1$ 作为列表的索引。

练习 3.9 (难)

定义不用循环的 `reverse` 方法。改为用你迄今为止开发过的方法。

提示

用到的方法是 `foldLeft` 和 `append`。一开始定义一个 `prepend` 方法可能会有用，它按照 `append` 来定义，往列表的最前面添加一个元素。

答案 3.9

你可以先定义一个 `prepend` 的函数式方法，它允许在列表前添加一个元素。可以通过左折叠列表来完成，用一个包含了待添加元素的累加器而非空列表：

```
public static <T> List<T> prepend(T t, List<T> list) {
    return foldLeft(list, list(t), a -> b -> append(a, b));
}
```

然后将 `reverse` 方法定义为左折叠，始于空列表并以 `prepend` 方法为操作：

```
public static <T> List<T> reverse(List<T> list) {
    return foldLeft(list, list(), x -> y -> prepend(y, x));
}
```

完成以后，你可以用对应的实现替换对 `prepend` 的调用：

```
public static <T> List<T> reverse(List<T> list) {
    return foldLeft(list, list(), x -> y ->
        foldLeft(x, list(y), a -> b -> append(a, b)));
}
```

警告 不要在生产代码中使用答案 3.9 中 `reverse` 和 `prepend` 的实现。

它们都悄悄地遍历了整个列表好几次，所以很慢。在第 5 章中，你将学习如何创建在所有场合都表现良好的函数式不可变列表。

练习 3.10 (难)

在 3.10 节中，你通过对每个元素应用一个操作定义了一个映射列表的方法。这个操作的实现包括了一个折叠。用 `foldLeft` 或 `foldRight` 重写 `map` 方法。

提示

要解决这个问题，应该使用刚定义的 `append` 或 `prepend` 方法。

答案 3.10

为了理解这个问题，你必须考虑到该映射包含了两个操作：将函数应用于每个元素，然后把所有元素收集到新列表中。第二个操作是折叠，其单位元为空列表（在静态导入 `CollectionUtilities.*` 后可以写成 `list()`），操作为向列表中添加一个元素。

以下是一个使用 `append` 和 `foldLeft` 方法的实现：

```
public static <T, U> List<U> mapViaFoldLeft(List<T> list,
                                           Function<T, U> f) {
    return foldLeft(list, list(), x -> y -> append(x, f.apply(y)));
}
```

以下实现使用 `foldRight` 和 `prepend`：

```
public static <T, U> List<U> mapViaFoldRight(List<T> list,
                                              Function<T, U> f) {
    return foldRight(list, list(), x -> y -> prepend(f.apply(x), y));
}
```

函数式编程之美有一部分就是寻找每个可以被抽象和重用的小元素。在习惯了这种思维方式之后，你就会开始看到，模式无所不在并且会想要抽象它们。

你可以通过复合刚才编写的基本列表函数来定义许多其他有用的函数。但是我们要把对它们的学习延迟到第5章，那时你将学到用纯函数式不可变列表来代替传统的Java列表能提供的诸多优点，包括对大多数函数式操作的更好性能。

3.3.6 复合映射和映射复合

将多次转换应用于列表元素并不罕见。想象有一个价格表，你打算对所有人应用9%的税，然后增加3.50美元的固定运费。可以通过复合两个映射来实现：

```
Function<Double, Double> addTax = x -> x * 1.09;
Function<Double, Double> addShipping = x -> x + 3.50;
List<Double> prices = list(10.10, 23.45, 32.07, 9.23);
List<Double> pricesIncludingTax = map(prices, addTax);
List<Double> pricesIncludingShipping =
    map(pricesIncludingTax, addShipping);
System.out.println(pricesIncludingShipping);
```

代码打印出如下信息：

```
[14.509, 29.0605, 38.4563000000000006, 13.5607]
```

它能够工作，但是效率不够高，因为应用了两次映射。你可以这样来获得同样的结果：

```
System.out.println(map(map(prices, addTax), addShipping));
```

但这样仍然映射了两次。一个更好的办法是复合函数而不是复合映射，或者换句话说，映射复合而不是复合映射：

```
System.out.println(map(prices, addShipping.compose(addTax)));
```

也许你喜欢更“自然”的书写顺序：

```
System.out.println(map(prices, addTax.andThen(addShipping)));
```

3.3.7 对列表应用作用

在上一个示例中，你打印出列表以验证结果。在真实的情况下，你可能会对列表中的每个元素应用更复杂的作用。例如，你可以在格式化为两位小数之后显示每个价格。这可以通过迭代来完成：

```
for (Double price : pricesIncludingShipping) {  
    System.out.printf("%.2f", price);  
    System.out.println();  
}
```

但是，你再次混合了可以被抽象的行为。可以像映射一样抽象迭代，而应用于每个元素的作用可以被抽象为类似函数般的东西，但是具有副作用并且没有返回值。这正是你在练习 3.1 的答案中使用的 Effect 接口。因此，本例可以重写为：

```
Effect<Double> printWith2decimals = x -> {  
    System.out.printf("%.2f", x);  
    System.out.println();  
};  
  
public static <T> void forEach(Collection<T> ts, Effect<T> e) {  
    for (T t : ts) e.apply(t);  
}  
  
forEach(pricesIncludingShipping, printWith2decimals);
```

代码看上去更多了，但是 Effect 接口和 forEach 方法可以只编写一次并重用，所以你可以独立测试它们。你的业务代码被缩减为仅一行。

3.3.8 处理函数式的输出

通过 `forEach` 方法，你多少可以抽象一些副作用。虽然你抽象了作用的应用以将其隔离，不过还可以走得更远。通过 `forEach` 方法，列表的每个元素都会应用一个作用。如果能够将这些作用复合成一个就好了。思考如何让它成为一个折叠并生成一个作用。如果能做到这一点，那么你的程序可以是一个完全函数式的程序，绝对不会有副作用。它将生成一个新程序，没有控制结构，只有应用一个个作用的列表。让我们开工吧！

为了表示程序的指令，你将使用清单 3.5 中用过的 `Executable` 接口。之后你需要一种方法来复合 `Executable` 的实例，可以通过一个函数式方法或一个函数来完成。既然你在函数式的语境中，我们还是用一个函数吧：

```
Function<Executable, Function<Executable, Executable>> compose =
    x -> y -> () -> {
        x.exec();
        y.exec();
    };

```

接下来，你需要一个中性元素或单位元，用于复合 `Executable`。这可比什么也不干的可执行代码还简单。让我们称它为 `ez`：

```
Executable ez = () -> {};

```

名称 `ez` 代表零可执行代码（`executable zero`），它表示由复合可执行代码所组成的操作的零（或标识）元素。

现在你可以按如下方式来编写纯函数式的程序：

```
Executable program = foldLeft(pricesIncludingShipping, ez,
    e -> d -> compose.apply(e).apply(() -> printWith2decimals.apply(d)));

```

看起来可能有点复杂，但其实很简单。它是对列表 `pricesIncludingShipping` 的一个 `foldLeft`，使用 `ez` 作为累加器的初始值。唯一稍微复杂的部分是函数。如果你忘了柯里化形式并把它想成了一个双参函数，那么可以认为它接收一个 `Executable(e)` 作为第一个参数以及一个 `Double(d)` 作为第二个参数，并且它复合了第一个参数和一个由 `printWith2decimals` 方法应用于 `Double` 所生成的新 `Executable`。如你所见，这只是一个复合抽象的问题！

请注意，你没有应用任何副作用。你得到的是一个用新语言编写的新程序（或

者说是一个脚本)。可以通过调用 `exec()` 来执行这个程序：

```
program.exec();
```

可以得到如下结果：

```
14.51
29.06
38.46
13.56
```

这让你初尝函数式编程如何无副作用地生成输出的滋味。你自己决定是否要在生产代码中使用这种技术。真正的函数式语言不会让你选择，但 Java 并不是函数式语言，所以你还是可以选择的。如果决定要函数式地编程，你可能会错过一些在这个领域中有助于你的东西，但重要的是要知道一切皆有可能。

3.3.9 构建反递归列表

程序员们一次又一次地构建反递归列表，其中的大部分都是整型列表。如果你觉得身为 Java 程序员的你并没有频繁这样做，那么请思考以下示例：

```
for (int i = 0; i < limit; i++) {
    some processing...
}
```

这段代码是两个抽象的复合：一个反递归列表和一些处理。反递归列表是从 0 (包含) 到 `limit` (除外) 的整型列表。正如我们已经提到的，函数式编程的诸多特点之一，就是将抽象推到极致。因此，让我们来抽象这个反递归列表的构造。

正如我先前所说，反递归意味着每个元素都可以从第一个元素开始，对上一个元素应用一个函数。以此来区分反递归和递归的构造。(在递归构造中，从最后一个元素开始，每个元素都是上一个元素的函数。) 我们将在第 4 章再回到这一点，但是现在，这意味着反递归列表很容易构造。只要从第一个元素 (`int i = 0`) 开始，应用所选择的函数 (`i -> i++`) 即可。

你可以先构建列表，再将其映射到与 `some processing...` 相对应的函数，或是映射到复合函数或作用。让我们用一个具体的上限：

```
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

这基本上等价于以下内容：

```
list(0, 1, 2, 3, 4).forEach(System.out::println);
```

你抽象了列表和作用。但是还可以进行进一步抽象。

练习 3.11

用一个起始值、一个上限和函数 $x \rightarrow x + 1$ 来编写一个生成列表的方法。你将称此方法为 `range`，它将具有以下签名：

```
List<Integer> range(int start, int end)
```

答案 3.11

你可以用 `for` 循环来实现 `range` 方法。但是你要用一个 `while` 循环来为下一个练习做准备：

```
public static List<Integer> range(int start, int end) {
    List<Integer> result = new ArrayList<>();
    int temp = start;
    while (temp < end) {
        result = CollectionUtilities.append(result, temp);
        temp = temp + 1;
    }
    return result;
}
```

我选了 `while` 循环，因为它更容易转换为可以应用于任何类型的通用方法，只要给定一个从该类型到其本身的函数和另一个从该类型到 `Boolean` 的函数（称为 `predicate`）即可。

练习 3.12

编写一个任何类型和任何条件都适用的通用 `range` 方法。因为区间（`range`）的概念主要用于数字，让我们称这个方法为 `unfold` 并赋予它以下签名：

```
List<T> unfold(T seed, Function<T, T> f, Function<T, Boolean> p)
```

答案 3.12

从 `range` 方法的实现开始，你要做的全部就是将特定的部分替换为通用的：

```
public static <T> List<T> unfold(T seed,
```

```

        Function<T, T> f,
        Function<T, Boolean> p) {
List<T> result = new ArrayList<>();
T temp = seed;
while (p.apply(temp)) {
    result = append(result, temp);
    temp = f.apply(temp);
}
return result;
}

```

练习 3.13

用 `unfold` 方法来实现 `range` 方法。

答案 3.13

这并没有什么难度。你必须提供 `range` 的 `start` 参数作为 `seed`；函数 `f` 为 `x -> x + 1`；还有断言 `p` 为 `x -> x < end`：

```

public static List<Integer> range(int start, int end) {
    return unfold(start, x -> x + 1, x -> x < end);
}

```

反递归和递归具有双重关系。它们相互对应，所以总是可以将一个递归过程变成一个反递归，反之亦然。这是第 4 章的主题，那时你将学习如何将递归过程变为一个反递归的过程。现在先让我们来反转这个过程。

练习 3.14

根据你在先前章节中定义的函数方法来编写 `range` 的递归版本。

提示

你需要的唯一方法是 `prepend`，虽然你也可以用不同的方法来选择其他实现。

答案 3.14

定义一个递归的实现非常简单。你只需将 `start` 参数前置到 `prepend` 方法中，使用相同的 `end` 参数，并将 `start` 参数替换为应用 `f` 函数的结果。做起来比说容易：

```

public static List<Integer> range(Integer start, Integer end) {
    return end <= start
        ? CollectionUtilities.list()
        : CollectionUtilities.prepend(start, range(start + 1, end));
}

```

应用 `range` 方法来获得与先前示例里的 `for` 循环相同的结果很简单：

```
for (int i = 0; i < 5; i++) {  
    System.out.println(i);  
}
```

你可以进行如下重写：

```
range(0, 5).forEach(System.out::println);
```

更有意思的是，如果在 `for` 循环中应用的过程是函数式的，那么收益将更加令人惊叹：

```
List<Integer> list = new ArrayList<>();  
for (int i = 0; i < 5; i++) {  
    list.add(i * i);  
}
```

可以进行如下替换（假设静态导入了 `CollectionUtilities.*`）：

```
mapViaFoldLeft(range(0, 5), x -> x * x);
```

当然，在这个例子中，也可以使用 `mapViaFoldRight`。

基于栈的递归的危害

在先前示例中开发的递归实现不应该被用于生产，因为它受限于 6000 到 7000 步之间。如果你试着再进一步，栈将会溢出。第 4 章提供了这个主题的更多信息。

严格带来的危害

这些版本（递归和反递归）都与 `for` 循环不等价。这是因为，虽然 Java 基本上算是一门严格的语言（在方法参数上是严格），但是 `for` 循环是惰性的，正如 Java 的所有控制结构和某些运算符。以 `for` 循环为例，这表示赋值的顺序将是索引、计算、索引、计算……而使用 `range` 方法将会在映射函数之前先计算出完整的列表。

这个问题的出现是因为不应该将列表用于此处：列表是严格的数据结构。但是你总得从什么地方开始。在第 9 章中，你将学习如何构建惰性集合，它可以解决这个问题。

在本节中，你学习了如何抽象和封装在使用诸如列表的命令式数据结构时不可避免的命令式操作。在第 5 章中，你还将学习如何用完全纯函数式的数据结构替换这些旧数据结构，它还能提供更多的自由与更佳的性能。同时，你必须更仔细地查看类型。

3.4 使用正确的类型

在前面的示例中，你用了诸如整数、双精度和字符串这些标准类型来表示价格和电子邮件地址等业务实体。虽然这是在命令式编程中的常见做法，但它会导致不该出现的问题。如我所言，你应该更信任类型而不是名称。

3.4.1 标准类型的问题

让我们来调查一个简化的问题，看看用标准类型解决会如何导致问题。假设你有产品的名称、价格和重量，需要创建代表产品销售的发票。这些发票上要提到产品、数量、总价格和总重量。

你可以用如下的 `Product` 类来表示一个产品：

```
public class Product {  
  
    private final String name;  
    private final double price;  
    private final double weight;  
  
    public Product(String name, double price, double weight) {  
        this.name = name;  
        this.price = price;  
        this.weight = weight;  
    }  
  
    ... (getters)  
}
```

由于属性为 `final`，需要一个构造函数来做初始化，还需要一些 `getter` 来读取它们，不过我们没有展示 `getter`。

接下来可以使用 `OrderLine` 类来表示订单的每一项。这个类在清单 3.10 中进行了显示。

清单 3.10 表示订单中某一项的组件

```
public class OrderLine {  
  
    private Product product;  
    private int count;  
  
    public OrderLine(Product product, int count) {  
        super();  
        this.product = product;  
    }  
}
```



```

        this.count = count;
    }

    public Product getProduct() {
        return product;
    }

    public void setProduct(Product product) {
        this.product = product;
    }

    public int getCount() {
        return count;
    }

    public void setCount(int count) {
        this.count = count;
    }

    public double getWeight() {
        return this.product.getWeight() * this.count;
    }

    public double getAmount() {
        return this.product.getPrice() * this.count;
    }
}

```

这看起来像是一个还不错的传统 Java 对象，由 Product 和 int 初始化，表示一个订单项。它还有计算总价格和总重量的方法。

由于决定继续使用标准类型，所以你将用 List<OrderLine> 来表示订单。清单 3.11 展示了如何处理订单。（如果你还不习惯函数式风格，可以对这段代码和与之等价的命令式代码 StoreImperative 进行比较，命令式代码可以在本书的网站 <https://github.com/fpinjava/fpinjava> 上找到）。

清单 3.11 处理订单

```

import java.util.List;
import static com.fpinjava.common.CollectionUtilities.*;

public class Store {

    public static void main(String[] args) {
        Product toothPaste = new Product("Tooth paste", 1.5, 0.5);
        Product toothBrush = new Product("Tooth brush", 3.5, 0.3);
        List<OrderLine> order = list(
            new OrderLine(toothPaste, 2),
            new OrderLine(toothBrush, 3));
    }
}

```

```
double weight = foldLeft(order, 0.0, x -> y -> x + y.getAmount());  
double price = foldLeft(order, 0.0, x -> y -> x + y.getWeight());  
System.out.println(String.format("Total price: %s", price));  
System.out.println(String.format("Total weight: %s", weight));  
}  
}
```

运行这段程序会在控制台上显示以下结果：

```
Total price: 1.9  
Total weight: 13.5
```

看上去还挺好，但是错了！问题在于编译器没有告诉你关于错误的任何信息。捕获这个错误的唯一办法就是测试程序，但测试并不能证明程序是正确的。它们只能证明你还不能通过编写另一段程序（顺便说一句，也可能是不正确的）来证明它不正确。

如果你没有注意到（这是不可能的），问题出在下面这几行：

```
double weight = foldLeft(order, 0.0, x -> y -> x + y.getAmount());  
double price = foldLeft(order, 0.0, x -> y -> x + y.getWeight());
```

你把价格和重量搞混了，而编译器无法发现，因为它们都是 `double` 类型的。

顺便说一句，如果学过建模，你可能会回想起一个老规则：类不应该有相同类型的多个属性。与此相反，它们应该有一个具有特定基数（cardinality）的属性。这意味着这里的一个产品应该有一个类型为 `double` 的属性，其基数为 2。这显然不是解决问题的正确方法，但它是一个值得记住的好规则。如果你发现自己正在用多个相同类型的属性建模对象，那你可能弄错了什么。

可以做什么来避免这样的问题呢？首先，你必须意识到价格和重量不是数字。它们是数量。数量可以是数字，但价格是货币单位的数量，而重量是重量单位的数量。你永远不应该对磅和美元求和。

3.4.2 定义值类型

应该使用值类型（value type）来避免这个问题。值类型是代表值的类型。你可以定义一个值类型来表示一个价格：

```
public class Price {  
  
    public final double value;
```

```
public Price(double value) {  
    this.value = value;  
}  
}
```

还可以为重量再来一次：

```
public class Weight {  
  
    public final double value;  
  
    public Weight(double value) {  
        this.value = value;  
    }  
}
```

但是这样并不能解决你的问题，因为代码还可以这样编写：

```
weight += orderLine.getAmount().value;  
price += orderLine.getWeight().value;
```

你需要通过一个方法来为 Price 和 Weight 定义加法：

```
public class Price {  
  
    ...  
  
    public Price add(Price that) {  
        return new Price(this.value + that.value);  
    }  
  
    ...  
}
```

你也需要乘法，但乘法略有不同。加法增加类型相同的东西，而乘法是将一种类型的东西乘以一个数字。所以乘法在不只是应用于数字时不是可交换的。下面是 Product 的乘法示例：

```
public Price mult(int count) {  
    return new Price(this.value * count);  
}
```

在程序中，你从零开始增加价格和重量。现在不能再这样做了，所以需要有一个零的 Price 和一个零的 Weight。它可以是一个单例，所以你将

```
public static final Price ZERO = new Price(0.0);
```

用于 Price 类，Weight 类也是一样。

Product 类需要进行如下修改：

```

public class Product {
    public final String name;
    public final Price price;
    public final Weight weight;

    public Product(String name, Price price, Weight weight) {
        this.name = name;
        this.price = price;
        this.weight = weight;
    }
}

```

OrderLine 也要改：

```

public Weight getWeight() {
    return this.product.getWeight().mult(this.count);
}

public Price getAmount() {
    return this.product.price.mult(this.count);
}

```

你现在可以使用这些类型和操作来重写程序：

```

import static com.fpinjava.common.CollectionUtilities.*;
import java.util.List;

public class Store {

    public static void main(String[] args) {

        Product toothPaste = new Product("Tooth paste", new Price(1.5), new Weight(0.5));
        Product toothBrush = new Product("Tooth brush", new Price(3.5), new Weight(0.3));

        List<OrderLine> order = list(
            new OrderLine(toothPaste, 2),
            new OrderLine(toothBrush, 3));

        Price price = Price.ZERO;
        Weight weight = Weight.ZERO;
        for (OrderLine orderLine : order) {
            price = price.add(orderLine.getAmount());
            weight = weight.add(orderLine.getWeight());
        }
    }
}

```

没有了编译器的警告，你也不会再混淆类型了。不过还可以做得更好。首先，你可以添加对 `Price` 和 `Weight` 的验证。除了类本身内部的单位元以外，它们都不应该通过一个零值来构造。你可以使用一个私有构造函数和一个工厂方法来实现。对 `Price` 的处理方法如下：

```
private Price(double value) {
    this.value = value;
}

public static Price price(double value) {
    if (value <= 0) {
        throw new IllegalArgumentException("Price must be greater than 0");
    } else {
        return new Price(value);
    }
}
```

但是你能做的主要改变是重用在 3.3 节中开发的折叠函数。这些函数接收一个函数作为它们的第三个参数，所以首先你需要为增加价格定义一个函数（在 `Price` 类中）：

```
public static Function<Price, Function<OrderLine, Price>> sum =
    x -> y -> x.add(y.getAmount());
```

你还需要在 `Weight` 类中使用相同的函数以增加重量：

```
public static Function<Weight, Function<OrderLine, Weight>> sum =
    x -> y -> x.add(y.getWeight());
```

最后在 `Price` 和 `Weight` 中添加一个 `toString` 方法，以简化测试：

```
public String toString() {
    return Double.toString(this.value);
}
```

现在你可以用折叠来修改 `Store` 类：

```
Product toothPaste = new Product("Tooth paste", price(1.5), weight(0.5));
Product toothBrush = new Product("Tooth brush", price(3.5), weight(0.3));
List<OrderLine> order =
    list(new OrderLine(toothPaste, 2), new OrderLine(toothBrush, 3));
Price price = foldLeft(order, Price.ZERO, Price.sum);
Weight weight = foldLeft(order, Weight.ZERO, Weight.sum);
System.out.println(String.format("Total price: %s", price));
System.out.println(String.format("Total weight: %s", weight));
```

3.4.3 值类型的未来

值类型可用于所有的业务类型，它给你的程序带来了类型安全。不过我所描述的值类型并不是真正的值类型。真正的值类型就如对象一般操作，如原始类型一般工作。其他语言有内置的值类型，但 Java 并没有，虽然这可能会改变；已经有了一个在未来版本的 Java 中包含值类型的提案。如果你对该主题感兴趣，可以访问 <http://cr.openjdk.java.net/~jrose/values/values-0.html> 来阅读该提案。

3.5 总结

- 通过确保没有在外界可见的状态变更，Java 的控制结构可以变得更加函数式。
- 控制结构可以从它们控制的作用中抽象出来。
- Result 接口可用于表示可能会失败的操作结果。
- 诸如 if...else 和 switch...case 等控制结构，可以被替换为函数。
- 迭代可以被抽象为函数，用以代替循环。
- 可以从两个方向（右和左）折叠列表，以将它们化简为一个对象（顺便说一句，它可能是一个新的列表）。
- 列表可以用递归或反递归处理。
- 列表可以映射函数，以更改其元素的值和 / 或类型。
- 映射可以通过折叠来实现。
- 可以在列表上绑定作用，以应用于其每个元素。
- 递归和反递归也可以用于构造列表。
- 递归深度受 Java 栈大小的限制。
- 通过允许编译器检测类型问题，使用值类型可以使程序更加安全。

4 递归、反递归和记忆化

本章要点

- 理解递归和反递归
- 使用递归函数
- 复合大量的函数
- 使用记忆化加速函数

第3章我们介绍了强大的方法和函数，但有些不应该在生产代码中使用，因为它们可能导致栈溢出并使应用程序（或至少是调用它们的线程）崩溃。这些“危险”的方法和函数主要是显式递归，但并不总是如此。你已经看到，复合函数也可能会栈溢出，甚至非递归函数也有可能发生，尽管这并不常见。

在本章中，你将学习如何将基于栈的函数转换为基于堆的函数。因为栈是一块有限的内存区域，所以很有必要这么做。为了让递归函数更加安全，你需要实现让它们使用堆（主存储器区域）以代替有限的栈空间。要完全理解这个问题，首先需要了解递归和反递归之间的差异。

4.1 理解反递归和递归

反递归从第一步开始，使用每一步的输出作为下一步的输入，以复合计算步骤。递归的操作相同，但它是从最后一步开始。在递归中，你必须延迟赋值，直至遇到基本条件（对应于反递归的第一步）为止。

假设你的编程语言只有两个指令：递增（值加 1）和递减（值减 1）。举例来说，你可以通过复合这些指令来实现求和。

4.1.1 探讨反递归和递归的加法例子

要对两个数字 x 和 y 求和，你可以执行以下操作：

- 如果 $y=0$ ，返回 x 。
- 否则，递增 x ，递减 y ，并重新开始。

可以用 Java 进行如下编写：

```
static int add(int x, int y) {  
    while(y > 0) {  
        x = ++x;  
        y = --y;  
    }  
    return x;  
}
```

这里还有一个更简单的办法：

```
static int add(int x, int y) {  
    while(y-- > 0) {  
        x = ++x;  
    }  
    return x;  
}
```

因为在 Java 中，所有参数都是通过值传递的，因此直接用参数 x 和 y 没问题。另外值得注意的是，你用了后缀自减运算（post-decrementation）来简化编码。可以通过稍微改变条件来使用前缀自减运算（pre-decrementation），从而将从 y 到 1 的迭代，切换为从 $y - 1$ 到 0 的迭代：

```
static int add(int x, int y) {  
    while(--y >= 0) {  
        x = ++x;  
    }
```

```
}  
    return x;  
}
```

递归版本更加麻烦一些，但仍然很简单：

```
static int addRec(int x, int y) {  
    return y == 0  
        ? x  
        : addRec(++x, --y);  
}
```

这两种方法似乎都能工作，但如果你试图用数字较大的递归版本，可能会大吃一惊。虽然这个版本，

```
addRec(10000, 3);
```

可以生成预期结果 10003，但是这样交换参数，

```
addRec(3, 10000);
```

则生成了一个 `StackOverflowException`。

4.1.2 在 Java 中实现递归

要了解发生了什么，你需要知道 Java 如何处理方法调用。当调用一个方法时，Java 暂停手上正在处理的事情，并将环境推入栈内，为执行被调用的方法提供一个位置。当该方法返回时，Java 弹出栈以恢复先前的环境并继续执行程序。如果你接二连三地调用方法，栈为这些方法总是最多持有一个调用环境。

但是复合方法不仅仅是接二连三的调用。方法调用方法。如果 `method1` 的部分实现是调用 `method2`，Java 就会再次暂停 `method1` 的执行，将当前环境推入栈内并开始执行 `method2`。当 `method2` 返回时，Java 从栈内弹出最后一个推入的环境，并继续执行（本例中是 `method1`）。当 `method1` 完成时，Java 再次从栈内弹出最后一个环境，并恢复它在调用该方法之前正在执行的操作。

方法调用可能被深度嵌套，而嵌套的深度有一个限制，那就是栈的大小。在当前情况下，这个限制大约是几千，并且可以通过配置栈的大小来增加这个限制值。但是由于所有线程都会使用相同的栈，增加栈的大小通常只是浪费空间。默认栈空间从 320 KB 到 1024 KB 不等，具体取决于 Java 版本和所用的系统。对于栈使用率最低的 64 位 Java 8 程序来说，嵌套方法调用的最大数量约为 7000。一般来说，除

了某些特定情况外，不需要更多。递归方法调用就是这么一种特定的情况。

4.1.3 使用尾调用消除

为了在方法调用返回之后恢复计算，一般来说将环境推入栈内很有必要，但并不总是如此。如果方法调用做的最后一件事是调用某一个方法，当此方法返回时并没有什么可以恢复，所以应该可以直接恢复为当前方法的调用者而不是当前方法自己。在尾部位置发生的方法调用，意味着它是返回之前要做的最后一件事，称为尾调用（tail call）。避免将环境推入栈内以在尾调用之后继续方法处理，是称为尾调用消除（tail call elimination, TCE）的优化技术。不幸的是，Java 不用 TCE。

尾调用消除有时被称为尾调用优化（tail call optimization, TCO）。一般来说，TCE 只是一个优化，没有它你也可以过得好好的。但是当涉及递归函数调用时，TCE 就不再只是优化了，而是一个必备特性。这就是为什么在处理递归时，TCE 这个术语比 TCO 更好。

4.1.4 使用尾递归方法和函数

大多数函数式编程语言都有 TCE。但是 TCE 并不足以应付每一个递归调用。要成为 TCE 的候选者，递归调用必须是方法所做的最后一件事。

思考以下计算列表元素的总和的方法：

```
static Integer sum(List<Integer> list) {  
    return list.isEmpty()  
        ? 0  
        : head(list) + sum(tail(list));  
}
```

这个方法使用了第 3 章中的 head 和 tail 方法。对 sum 方法的递归调用并不是方法做的最后一件事情。该方法执行的最后 4 件事情如下：

- 调用 head 方法
- 调用 tail 方法
- 调用 sum 方法
- 对 head 的结果和 sum 的结果求和

即使有 TCE，你也不能用这个方法来应对 10,000 个元素的列表。但是你可以重

写这个方法以使 `sum` 在尾部位置调用：

```
static Integer sum(List<Integer> list) {  
    return sumTail(list, 0);  
}  
  
static Integer sumTail(List<Integer> list, int acc) {  
    return list.isEmpty()  
        ? acc  
        : sumTail(tail(list), acc + head(list));  
}
```

这里的 `sumTail` 方法是尾递归，可以通过 TCE 进行优化。

4.1.5 抽象递归

到目前还好，但是如果 Java 没有 TCE，为什么要搞得这么麻烦？嗯，虽然 Java 没有 TCE，但是你也可以应付。需要做的全部事情如下：

- 表示未计算的方法调用。
- 将它们存储在栈式结构中，直至满足终止条件。
- 以“后进先出”（LIFO）的顺序对计算进行调用。

递归方法的大部分例子都是阶乘函数，要不就是斐波那契数列。除了递归之外，阶乘方法实在没什么意思。斐波那契数列会更有意思一些，我们稍后再回来。首先，你将使用本章开头所示的简单版递归加法。

递归和反递归函数都是函数，其中 $f(n)$ 是 $f(n-1)$ 、 $f(n-2)$ 和 $f(n-3)$ 等的复合，直至遇上终止条件（一般是 $f(0)$ 或 $f(1)$ ）。记住，在传统编程中，复合通常意味着复合计算结果。也就是说，复合函数 $f(a)$ 和 $g(a)$ 包括对 $g(a)$ 求值，然后使用结果作为 f 的输入。但其实不必这样做。在第2章中，你开发了一个 `compose` 方法用于复合函数，还有一个 `higherCompose` 函数也可以同样使用。不是必须要计算复合函数，它们只不过生成了可以之后再应用的另一个函数罢了。

递归和反递归非常相似，但是有一个不同之处。你可以创建一个函数调用列表，而不是函数列表。使用反递归，每一步都是终止的，因此可以对它求值以获得结果，并使其成为下一步的输入。使用递归，你从另一端开始，所以必须将未求值的调用放在列表中，直到找到终止条件为止，然后就可以反序处理列表。将这些步骤入栈，直至找到最后一个，然后反序（后进先出）处理栈，再次计算每一个步骤，并用结

果作为下一个（实际上是上一个）函数的输入。

问题是 Java 对递归和反递归都用了线程栈，并且其容量有限。通常，栈会在 6000 到 7000 步之后溢出。你要做的是创建一个返回未计算步骤的函数或方法。为了表示计算中的一个步骤，你要用一个名为 `TailCall` 的抽象类（因为你想要表示调用出现在尾部位置的方法）。

这个 `TailCall` 抽象类有两个子类。一个表示中间调用，即暂停某一步的处理以再次调用该方法对下一步求值。它由一个名为 `Suspend` 的子类表示。它实例化了 `Supplier<TailCall>>`，表示下一个递归调用。这样的话，不必将所有的 `TailCall` 都放在一个列表中，你可以用把每个尾调用链接到下一个的办法来构造一个链表。这个办法的好处在于链表是一个栈，提供常数级别的插入时间以及对最后插入的元素的常数级访问时间，它是对 LIFO 结构的优化。

第二个子类表示最后一个调用，它应该返回结果，所以称它为 `Return`。它不会持有到下一个 `TailCall` 的连接，因为并没有下一个，但是它将持有结果。你将得到如下所示的程序：

```
public abstract class TailCall<T> {
    public static class Return<T> extends TailCall<T> {
        private final T t;
        public Return(T t) {
            this.t = t;
        }
    }

    public static class Suspend<T> extends TailCall<T> {
        private final Supplier<TailCall<T>> resume;
        private Suspend(Supplier<TailCall<T>> resume) {
            this.resume = resume;
        }
    }
}
```

你需要一些方法来处理这些类：一个返回结果，一个返回下一个调用，还有一个决定 `TailCall` 是 `Suspend` 还是 `Return` 的辅助方法。可以不用最后一个方法，但这样你就不得不使用丑陋的 `instanceof` 来做这件事。这三个方法如下所示：

```
public abstract TailCall<T> resume();
public abstract T eval();
public abstract boolean isSuspend();
```

`Return` 并没有实现 `resume` 方法，并且会抛出运行时异常。你的 API 的用户

不应该在同一个情况下调用这个方法，所以如果最后它被调用了，那就是一个 bug，你要停止应用程序。在 Suspend 类中，这个方法将会返回下一个 TailCall。

eval 方法返回存储在 Return 类中的结果。在第一个版本中，如果在 Suspend 类上调用，它将抛出一个运行时异常。

isSuspend 方法在 Suspend 中返回 true，在 Return 中返回 false。清单 4.1 展示了第一个版本。

清单 4.1 TailCall 接口和它的两个实现

```
public abstract class TailCall<T> {

    public abstract TailCall<T> resume();
    public abstract T eval();
    public abstract boolean isSuspend();

    public static class Return<T> extends TailCall<T> {

        private final T t;

        public Return(T t) {
            this.t = t;
        }

        @Override
        public T eval() {
            return t;
        }

        @Override
        public boolean isSuspend() {
            return false;
        }

        @Override
        public TailCall<T> resume() {
            throw new IllegalStateException("Return has no resume");
        }
    }

    public static class Suspend<T> extends TailCall<T> {

        private final Supplier<TailCall<T>> resume;

        public Suspend(Supplier<TailCall<T>> resume) {
            this.resume = resume;
        }
    }
}
```

```

@Override
public T eval() {
    throw new IllegalStateException("Suspend has no value");
}

@Override
public boolean isSuspend() {
    return true;
}

@Override
public TailCall<T> resume() {
    return resume.get();
}
}

```

为了让递归方法 `add` 能够用于任意数量的步骤上（在可用的内存限制之内），需要做一些更改。从你原来的方法开始，

```

static int add(int x, int y) {
    return y == 0
        ? x
        : add(++x, --y);
}

```

你需要按清单 4.2 中展示的进行修改。

清单 4.2 修改后的递归方法

```

static TailCall<Integer> add(int x, int y) {
    return y == 0
        ? new TailCall.Return<>(x)
        : new TailCall.Suspend<>(() -> add(x + 1, y - 1));
}

```

③ 在非终止条件下，返回一个 `Suspend`

在终止条件下，返回 `Return` ②

① 方法返回一个 `TailCall`

该方法返回一个 `TailCall<Integer>` 而不是一个 `int` ①。如果满足终止条件 ②，则返回值为 `Return<Integer>`，如果不满足终止条件 ③，则返回值为 `Suspend<Integer>`。`Return` 使用计算结果（因为 `y` 为 0，所以结果为 `x`）来实例化，而 `Suspend` 用 `Supplier<TailCall<Integer>>` 来实例化，它在执行顺序上是计算的下一步，亦是调用顺序的上一步。重要的是要理解 `Return` 对应于方法调用的最后一步，亦对应于计算的第一步。还要注意的，我们稍微改变了计算，将 `++x` 和 `--y` 替换为 `x + 1` 和 `y - 1`。必须这么做，因为我们用了一个闭包，只有当其

封闭的变量为 `final` 时才能工作。这算是作弊，但还能忍受。我们也可以创建和调用使用原始运算符的 `dec` 和 `inc` 这两个方法。

该方法返回一个 `TailCall` 的实例链，所有的实例都是 `Suspend`，除了最后一个为 `Return`。

到目前还好，但是这个方法并不能直接替换原来的方法。这没什么大不了的！原来的方法可像下面这样使用：

```
System.out.println(add(x, y))
```

你可以像下面这样使用这个新方法：

```
TailCall<Integer> tailCall = add(3, 100000000);
while(tailCall.isSuspend()) {
    tailCall = tailCall.resume();
}
System.out.println(tailCall.eval());
```

它看起来不好吗？如果让你感到失望，我能理解。你认为你应该只是透明地用一个新方法替换旧方法。你似乎还离得很远，但是再努力一些，就能做得更好。

4.1.6 为基于栈的递归方法使用一个直接替代品

在上一节的开始我曾说过，使用你递归 API 的用户没有机会通过调用 `Return` 的 `resume` 或是 `Suspend` 的 `eval` 来把 `TailCall` 实例搞砸。这很容易就能通过将计算代码放在 `Suspend` 类的 `eval` 方法中来实现：

```
public static class Suspend<T> extends TailCall<T> {
```

```
...
```

```
@Override
public T eval() {
    TailCall<T> tailRec = this;
    while(tailRec.isSuspend()) {
        tailRec = tailRec.resume();
    }
    return tailRec.eval();
}
```

你现在可以用更简单并更安全的方式来获取递归调用的结果了：

```
add(3, 100000000).eval()
```

但这不是你想要的，你不想调用 `eval` 方法。可以通过一个辅助方法来完成：

```
public static int add(int x, int y) {
    return addRec(x, y).eval();
}

private static TailCall<Integer> addRec(int x, int y) {
    return y == 0
        ? ret(x)
        : sus(() -> addRec(x + 1, y - 1));
}
```

现在你可以像原来那样调用 `add` 方法了。可以通过用静态工厂方法实例化 `Return` 和 `Suspend` 来使你的递归 API 更容易使用，这也允许你将 `Return` 和 `Suspend` 内部子类设置为私有类型：

```
public static <T> Return<T> ret(T t) {
    return new Return<>(t);
}

public static <T> Suspend<T> sus(Supplier<TailCall<T>> s) {
    return new Suspend<>(s);
}
```

清单 4.3 展示了完整的 `TailCall` 类。它增加了一个私有的无参构造函数来防止被其他类继承。

清单 4.3 完整的 `TailCall` 类

```
public abstract class TailCall<T> {

    public abstract TailCall<T> resume();
    public abstract T eval();
    public abstract boolean isSuspend();

    private TailCall() {}

    private static class Return<T> extends TailCall<T> {

        private final T t;

        private Return(T t) {
            this.t = t;
        }

        @Override
        public T eval() {
            return t;
        }
    }
}
```

```

    }

    @Override
    public boolean isSuspend() {
        return false;
    }

    @Override
    public TailCall<T> resume() {
        throw new IllegalStateException("Return has no resume");
    }
}

private static class Suspend<T> extends TailCall<T> {

    private final Supplier<TailCall<T>> resume;

    private Suspend(Supplier<TailCall<T>> resume) {
        this.resume = resume;
    }

    @Override
    public T eval() {
        TailCall<T> tailRec = this;
        while(tailRec.isSuspend()) {
            tailRec = tailRec.resume();
        }
        return tailRec.eval();
    }

    @Override
    public boolean isSuspend() {
        return true;
    }

    @Override
    public TailCall<T> resume() {
        return resume.get();
    }
}

public static <T> Return<T> ret(T t) {
    return new Return<>(t);
}

public static <T> Suspend<T> sus(Supplier<TailCall<T>> s) {
    return new Suspend<>(s);
}
}

```

现在你有了一个栈安全的尾递归方法，你能用一个函数再来一次吗？

4.2 使用递归函数

理论上，如果按照匿名类中的方法来实现函数，创建递归函数不应该比创建递归方法更难。但是 `lambda` 并没有实现为匿名类中的方法。

第一个问题是，`lambda` 在理论上无法递归。但这只是理论。事实上，你在第 2 章学到了绕过这个问题的一个技巧。静态定义的递归版 `add` 函数如下所示：

```
static Function<Integer, Function<Integer, TailCall<Integer>>> add =
    a -> b -> b == 0
        ? ret(a)
        : sus(() -> ContainingClass.add.apply(a + 1).apply(b - 1));
```

`ContainingClass` 在此代表定义函数的类名。与静态函数相比，你也许会更喜欢一个实例函数：

```
Function<Integer, Function<Integer, TailCall<Integer>>> add =
    a -> b -> b == 0
        ? ret(a)
        : sus(() -> this.add.apply(a + 1).apply(b - 1));
```

但是，你在此有着与 `add` 方法相同的问题，必须在结果上调用 `eval`。可以使用辅助方法加上递归实现的相同技巧，但你更应该让整件事情能够自包含。在其他语言中，例如 `Scala`，你可以在主函数内部定义局部辅助函数。在 `Java` 中可以吗？

4.2.1 使用局部定义的函数

在 `Java` 中不能直接在函数内部定义函数。但是写成 `lambda` 的函数是一个类。你能在该类中定义一个局部函数吗？实际上不能。你不能使用静态函数，因为局部类不能拥有静态成员，它们没有名字。你能使用实例函数吗？不能，因为需要一个对 `this` 的引用。`lambda` 和匿名类的区别之一就是 `this` 引用。在 `lambda` 中使用的 `this` 引用指向的是外围实例，而非指向匿名类实例。

解决办法是声明一个包含实例函数的局部类，如清单 4.4 所示。

清单 4.4 一个独立的尾递归函数

```
static Function<Integer, Function<Integer, Integer>> add = x -> y -> {
    class AddHelper {
        Function<Integer, Function<Integer, TailCall<Integer>>> addHelper =
            a -> b -> b == 0
                ? ret(a)
```

```

        : sus() -> this.addHelper.apply(a + 1).apply(b - 1));
    }
    return new AddHelper().addHelper.apply(x).apply(y).eval();
};

```

这里 this 的引用指的是 AddHelper 类

可以正常使用这个函数：

```
add.apply(3).apply(1000000000)
```

4.2.2 使函数成为尾递归

先前我说过，由于对列表中元素求和的一个简单的函数式递归方法不是尾递归，因此不能安全地处理它：

```

static Integer sum(List<Integer> list) {
    return list.isEmpty()
        ? 0
        : head(list) + sum(tail(list));
}

```

你也看到了不得不像下面这样改变方法：

```

static Integer sum(List<Integer> list) {
    return sumTail(list, 0);
}

static Integer sumTail(List<Integer> list, int acc) {
    return list.isEmpty()
        ? acc
        : sumTail(tail(list), acc + head(list));
}

```

原则很简单，就是有时候难以应用。它包括了使用持有计算结果的累加器。这个累加器被添加到方法的参数中。然后函数被转换为一个辅助方法，由原来的函数传入累加器的初值来调用。重要的是让这个过程更加自然，因为每当你编写一个递归方法或函数时都需要使用。

可以将一个方法变成两个方法。毕竟方法又不会旅行，所以你只需要使主方法公有、辅助方法（真正做事的方法）私有即可。对于函数同样如此，因为主函数对辅助函数的调用是一个闭包。局部定义的辅助函数优先于私有辅助方法的主要原因是避免命名冲突。

在允许局部定义函数的语言中，使用一个名称调用所有的辅助函数，是现今的一个实践，例如 go 或 process。这不能用非局部函数完成（除非你的每个类中只

有一个函数)。在上面的示例中，sum 的辅助函数被称为 sumTail。另一个现今的实践是调用与主函数同名的辅助函数，并加上下画线，如 sum_。无论你选择什么体系，最好保持一致。在本书的剩余部分，我将使用下画线来表示尾递归辅助函数。

4.2.3 双递归函数：斐波那契数列示例

任何关于递归函数的书都无法避免斐波那契数列函数。虽然它对我们大多数人来说完全没有用处，但它随处可见并且还很有趣。如果你从来都没有遇到这个函数的话，那让我们从需求开始。

斐波那契数列是一组数字，每个数字是前两个数字之和。这是一个递归定义。你需要一个终止条件，所以完整的需求如下所示：

- $f(0) = 0$
- $f(1) = 1$
- $f(n) = f(n-1) + f(n-2)$

这不是最初的斐波那契数列，最初的前两个数字等于 1。每个数字都应该是它在数列中的位置的函数，并且该位置从 1 开始。在计算中，你一般会喜欢从 0 开始。不过无论如何，问题不会改变。

为什么这个函数如此有趣？暂时先不回答这个问题，让我们试着来编写一个简单的实现：

```
public static int fibonacci(int number) {  
    if (number == 0 || number == 1) {  
        return number;  
    }  
    return fibonacci(number - 1) + fibonacci(number - 2);  
}
```

现在让我们编写一个简单的程序来测试这个方法：

```
public static void main(String args[]) {  
    int n = 10;  
    for(int i = 0; i <= n; i++){  
        System.out.print(fibonacci(i) + " ");  
    }  
}
```

如果运行这段测试程序，你就会得到前 10（或 9，取决于初始的定义）个斐波

那契数字：

0 1 1 2 3 5 8 13 21 34 55

基于你对 Java 的简单递归 (naive recursion) 的认知, 你也许会认为这个方法将成功计算 $f(n)$, 在栈溢出之前, n 能达到 6000 到 7000。好吧, 让我们检查一下。用 `int n = 6000` 替换 `int n = 10`, 看看会发生什么。启动程序, 然后可以喝杯咖啡。当你回来时, 你可以看到程序仍然在运行。它大约会达到 1 836 311 903 (你的数可能会不一样, 甚至会得到一个负数), 但它永远不会完成。没有栈溢出, 没有异常, 只是一直在运行着。究竟发生了什么事?

问题在于每次调用函数时都会创建两个递归调用。所以要计算 $f(n)$, 你需要 $2n$ 个递归调用。假设你的方法需要 10 纳秒来执行。(只是拍脑袋随便猜了一个值, 但你很快就会看到其实并没有什么影响。) 计算 $f(5000)$ 将需要 $2^{5000} \times 10$ 纳秒。你知道这有多久吗? 这个程序永远不会终止, 因为它需要的时间比太阳系的预期寿命还长 (甚至是宇宙)。

要创建可用的斐波那契函数, 必须将其更改为用单一的尾递归调用。还有另外一个问题: 结果是如此之大, 很快就会算术溢出, 以至于得到一个负数。

练习 4.1

创建一个尾递归版本的斐波那契函数式方法。

提示

正确的解决方案是使用累加器。但是由于有两个递归调用, 所以你需要两个累加器。

答案 4.1

让我们先编写辅助方法的签名。它接收两个作为累加器的 `BigInteger` 实例, 还有一个原来的参数, 并返回一个 `BigInteger`:

```
private static BigInteger fib_(BigInteger acc1, BigInteger acc2,
                               BigInteger x) {
```

你需要处理终止条件。如果参数为 0, 则返回 0:

```
private static BigInteger fib_(BigInteger acc1, BigInteger acc2,
                               BigInteger x) {
```

```
if (x.equals(BigInteger.ZERO)) {  
    return BigInteger.ZERO;  
}
```

如果参数为 1，则返回两个累加器之和：

```
private static BigInteger fib_(BigInteger acc1, BigInteger acc2,  
                               BigInteger x) {  
    if (x.equals(BigInteger.ZERO)) {  
        return BigInteger.ZERO;  
    } else if (x.equals(BigInteger.ONE)) {  
        return acc1.add(acc2);  
    }  
}
```

最后要处理递归。你必须执行以下操作：

- 接收累加器 2，并使其成为累加器 1。
- 通过对两个上一步的累加器求和来创建新的累加器 2。
- 把参数减 1。
- 递归调用函数并以这三个计算后的值为参数。

合并之后的代码如下：

```
private static BigInteger fib_(BigInteger acc1, BigInteger acc2,  
                               BigInteger x) {  
    if (x.equals(BigInteger.ZERO)) {  
        return BigInteger.ZERO;  
    } else if (x.equals(BigInteger.ONE)) {  
        return acc1.add(acc2);  
    } else {  
        return fib_(acc2, acc1.add(acc2), x.subtract(BigInteger.ONE));  
    }  
}
```

最后要做的是创建主方法，它调用这个辅助方法并传入累加器的初始值：

```
public static BigInteger fib(int x) {  
    return fib_(BigInteger.ONE, BigInteger.ZERO, BigInteger.valueOf(x));  
}
```

这只是实现的一个方案。你可以用稍微不同的方式来组织累加器、初始值和条件，只要它能工作就行。现在你可以调用 `fib(5000)`，它会在几纳秒内就给出结果。好吧，它需要几十毫秒，但这只是因为打印到控制台是一个缓慢的操作。我们很快就会回来。

无论是计算结果（1,045 位数字！）还是由于将双递归调用转换为单递归调用所增加的速度，都令人刮目相看。但是你仍然不能用这个方法处理 7500 以上的值。

练习 4.2

将这个方法转换为栈安全的递归方法。

答案 4.2

这应该很容易。以下代码展示了所需的修改：

```
BigInteger fib(int x) {
    return fib_(BigInteger.ONE, BigInteger.ZERO,
                BigInteger.valueOf(x)).eval();
}

TailCall<BigInteger> fib_(BigInteger acc1, BigInteger acc2, BigInteger x) {
    if (x.equals(BigInteger.ZERO)) {
        return ret(BigInteger.ZERO);
    } else if (x.equals(BigInteger.ONE)) {
        return ret(acc1.add(acc2));
    } else {
        return sus(() -> fib_(acc2, acc1.add(acc2), x.subtract(BigInteger.
ONE)));
    }
}
```

你现在可以计算 `fib(10000)`，然后数一数结果的位数！

4.2.4 让列表的方法变成栈安全的递归

在第3章中，你开发了用于列表的函数式方法。其中的一些方法是简单递归，因而不能用于生产环境。现在到了解决这个问题的时候了。

练习 4.3

创建 `foldLeft` 方法的栈安全递归版本。

答案 4.3

`foldLeft` 方法的简单递归版本是尾递归：

```
public static <T, U> U foldLeft(List<T> ts, U identity,
                               Function<U, Function<T, U>> f) {
    return ts.isEmpty()
        ? identity
        : foldLeft(tail(ts), f.apply(identity).apply(head(ts)), f);
}
```

可以很容易地把它转换成完全递归的方法：

```

public static <T, U> U foldLeft(List<T> ts, U identity,
                                Function<U, Function<T, U>> f) {
    return foldLeft_(ts, identity, f).eval();
}

private static <T, U> TailCall<U> foldLeft_(List<T> ts, U identity,
                                             Function<U, Function<T, U>> f) {
    return ts.isEmpty()
        ? ret(identity)
        : sus(() -> foldLeft_(tail(ts),
                                f.apply(identity).apply(head(ts)), f));
}

```

练习 4.4

创建 range 递归方法的完全递归版本。

提示

注意构造列表的方向（后置或前置）。

答案 4.4

range 方法不是尾递归：

```

public static List<Integer> range(Integer start, Integer end) {
    return end <= start
        ? list()
        : prepend(start, range(start + 1, end));
}

```

首先需要使用累加器来创建一个尾递归版本。你需要在这里返回一个列表，因此累加器就是一个列表，一开始要传入一个空列表。但是必须以相反的顺序构建列表：

```

public static List<Integer> range(List<Integer> acc,
                                   Integer start, Integer end) {
    return end <= start
        ? acc
        : range(append(acc, start), start + 1, end);
}

```

然后你需要通过使用真正的递归将此方法转换为一个主方法和一个辅助方法：

```

public static List<Integer> range(Integer start, Integer end) {
    return range_(list(), start, end).eval();
}

```

```
private static TailCall<List<Integer>> range_(List<Integer> acc,
                                             Integer start, Integer end) {
    return end <= start
        ? ret(acc)
        : sus(() -> range_(append(acc, start), start + 1, end));
}
```

事实上，你需要的反转操作是非常重要的。知道为什么吗？如果不知道，请尝试下一个练习。

练习 4.5 (难)

创建 `foldRight` 方法的栈安全递归版本。

答案 4.5

`foldRight` 方法基于栈的递归版本，如下所示：

```
public static <T, U> U foldRight(List<T> ts, U identity,
                                Function<T, Function<U, U>> f) {
    return ts.isEmpty()
        ? identity
        : f.apply(head(ts)).apply(foldRight(tail(ts), identity, f));
}
```

这个方法不是尾递归，所以我们先来创建一个尾递归版本。你可能会得到这样的结果：

```
public static <T, U> U foldRight(U acc, List<T> ts, U identity,
                                Function<T, Function<U, U>> f) {
    return ts.isEmpty()
        ? acc
        : foldRight(f.apply(head(ts)).apply(acc), tail(ts), identity, f);
}
```

不幸的是，这行不通！知道为什么吗？如果不知道，请测试该版本并比较标准版本的结果。你可以用上一章中设计的测试来比较这两个版本：

```
public static String addIS(Integer i, String s) {
    return "(" + i + " + " + s + ")";
}

List<Integer> list = list(1, 2, 3, 4, 5);
System.out.println(foldRight(list, "0", x -> y -> addIS(x, y)));
System.out.println(foldRightTail("0", list, "0", x -> y -> addIS(x, y)));
```

你将得到以下结果：

```
(1 + (2 + (3 + (4 + (5 + 0))))))  
(5 + (4 + (3 + (2 + (1 + 0))))))
```

这表明了列表以相反的顺序处理。一个简单的办法是在调用辅助方法之前在主方法中反转列表。如果在使方法栈安全和递归时用了这个技巧，你会得到以下结果：

```
public static <T, U> U foldRight(List<T> ts, U identity,  
                                Function<T, Function<U, U>> f) {  
    return foldRight_(identity, reverse(ts), f).eval();  
}  
  
private static <T, U> TailCall<U> foldRight_(U acc, List<T> ts,  
                                              Function<T, Function<U, U>> f) {  
    return ts.isEmpty()  
        ? ret(acc)  
        : sus(() -> foldRight_(f.apply(head(ts)).apply(acc), tail(ts), f));  
}
```

在第5章中，你将会通过 `foldRight` 实现 `foldLeft`，并通过 `foldLeft` 实现 `foldRight`，以此来开发反转列表的流程。因为 `reverse` 是一个 $O(n)$ 操作：由于需要遍历列表，执行它所需的时间与列表中的元素数量成正比，因此 `foldRight` 的递归实现性能不会太好。反转列表操作，由于遍历了列表两次而使耗时加倍。结论就是，当考虑使用 `foldRight` 时，你应该在以下操作中任选其一：

- 无视性能
- 修改函数（如果可能的话）以使用 `foldLeft`
- 仅对小列表使用 `foldRight`
- 使用命令式实现

4.3 复合大量函数

在第2章中，你见过如果试图复合大量的函数，就会栈溢出。原因与递归相同：因为复合函数导致方法调用方法。

需要复合 7000 多个函数可能不是你所期待的事。但是另一方面，没有理由不去实现它。如果可能的话，最终会有人用它找到一些有用的东西。如果它没有用，肯定会有人用它找到一些有趣的东西。

练习 4.6

编写一个函数 `composeAll`，接收从 `T` 到 `T` 的函数列表为参数，并返回复合列表中所有函数的结果。

答案 4.6

为了得到想要的结果，你可以使用一个右折叠，以函数列表、`identity` 函数（通过调用静态导入的 `Function.identity()` 方法获得），以及在第 2 章中编写的 `compose` 方法为参数：

```
static <T> Function<T, T> composeAll(List<Function<T, T>> list) {
    return foldRight(list, identity(), x -> y -> x.compose(y));
}
```

为了测试这个方法，你可以静态导入 `CollectionUtilities` 类（在第 3 章中开发过）中的所有方法，并像下面这样进行编写：

```
Function<Integer, Integer> add = y -> y + 1;
System.out.println(composeAll(map(range(0, 500), x -> add)).apply(0));
```

如果你不喜欢这种类型的代码，它等价于以下更易读的代码：

```
List<Function<Integer, Integer>> list = new ArrayList<>();
for (int i = 0; i < 500; i++) {
    list.add(x -> x + 1);
}

int result = composeAll(list).apply(0);
System.out.println(result);
```

运行这段代码显示 500，因为它是复合 500 个对参数加 1 的函数的结果。如果将 500 替换为 10 000 会怎样？你会得到一个 `StackOverflowException`。原因显而易见。

顺便说一句，在我测试这段代码的机器上，程序在接收 2856 个函数的列表时就崩溃了。

练习 4.7

解决了这个问题你就能复合（几乎）数量无限的函数。

答案 4.7

这个问题的解决办法很简单。你必须一直在更高的层次上复合这些函数的结果，

而不是嵌套调用。这意味着在每次函数调用之间，你都要返回到原调用处。如果这还不够清楚，想象这样做的命令式方法：

```
T y = identity;
for (Function<T, T> f : list) {
    y = f.apply(y);
}
```

这里的 `identity` 表示给定函数的单位元。这并不是复合函数，而是复合函数的应用。在循环结束时，你会得到一个 `T` 而不是一个 `Function<T, T>`。但这很容易解决。你可以创建一个从 `T` 到 `T` 的函数，它的实现如下：

```
static <T> Function<T, T> composeAll(List<Function<T, T>> list) {
    return x -> {
        T y = x;
        for (Function<T, T> f : list) {
            y = f.apply(y);
        }
        return y;
    };
}
```

← 创建了一份 `x` 的副本；你无法修改 `x`，因为它实际上必须为 `final`

你不能直接使用 `x`，因为它会创建一个闭包，所以它实际上应该是 `final`。这就是为什么你要创建一个副本。除了两件事以外，这段代码工作正常。

第一件事是它看起来并不像函数式，这可以通过使用折叠来轻松搞定。可以是左折叠或右折叠：

```
<T> Function<T, T> composeAllViaFoldLeft(List<Function<T, T>> list) {
    return x -> foldLeft(list, x, a -> b -> b.apply(a));
}
```

```
<T> Function<T, T> composeAllViaFoldRight(List<Function<T, T>> list) {
    return x -> foldRight(list, x, a -> b -> a.apply(b));
}
```

你用的是 `composeAllViaFoldRight` 实现的方法引用。这等价于以下代码：

```
<T> Function<T, T> composeAllViaFoldRight(List<Function<T, T>> list) {
    return x -> FoldRight.foldRight(list, x, a -> b -> a.apply(b));
}
```

如果你难以理解它是怎么工作的，想想与 `sum` 的相似之处。当定义 `sum` 时，列表是一个整型列表。初始值（这里为 `x`）为 0；`a` 和 `b` 是加法的两个参数；加法定义为 `a + b`。这里的列表是一个函数列表；初始值是恒等函数；`a` 和 `b` 是函数；实

现定义为 `b.apply(a)` 或 `a.apply(b)`。在 `foldLeft` 的版本中, `b` 是来自列表的函数, `a` 是当前结果。在 `foldRight` 的版本中, `a` 是来自列表的函数, `b` 是当前结果。

想要动手看到结果, 请参阅本书网站 (<https://github.com/fpinjava/fpinjava>) 上的代码中提供的可用的单元测试。

练习 4.8

代码有两个问题, 刚刚修复了一个。你能看到另一个问题并解决它吗?

提示

第二个问题在结果中看不到, 因为你复合的函数是特定的。事实上, 它们都是一个从整型到整型的函数。它们被复合的顺序无关紧要。尝试使用 `composeAll` 方法复合以下函数列表:

```
Function<String, String> f1 = x -> "(a" + x + ")";
Function<String, String> f2 = x -> "{b" + x + "}";
Function<String, String> f3 = x -> "[c" + x + "]";
System.out.println(composeAllViaFoldLeft(list(f1, f2, f3)).apply("x"));
System.out.println(composeAllViaFoldRight(list(f1, f2, f3)).apply("x"));
```

答案 4.8

我们已经实现了 `andThenAll` 而不是 `composeAll`! 为了得到正确的结果, 首先需要反转列表:

```
<T> Function<T, T> composeAllViaFoldLeft(List<Function<T, T>> list) {
    return x -> foldLeft(reverse(list), x, a -> b -> b.apply(a));
}

<T> Function<T, T> composeAllViaFoldRight(List<Function<T, T>> list) {
    return x -> foldRight(list, x, a -> a::apply);
}

<T> Function<T, T> andThenAllViaFoldLeft(List<Function<T, T>> list) {
    return x -> foldLeft(list, x, a -> b -> b.apply(a));
}

<T> Function<T, T> andThenAllViaFoldRight(List<Function<T, T>> list) {
    return x -> foldRight(reverse(list), x, a -> a::apply);
}
```

4.4 使用记忆化

在 4.2.3 节中，你实现了用一个函数来显示斐波那契数列。斐波那契数列的这个实现有一个问题：你想打印表示最高为 $f(n)$ 的数列的字符串，也就是说必须计算 $f(1)$ 、 $f(2)$ 等，直到 $f(n)$ 。但是要计算 $f(n)$ ，就必须递归地计算所有前面的值。最终，为了创建到 n 的数列，你将计算 n 次 $f(1)$ ， $n-1$ 次 $f(2)$ ，依此类推。计算的总次数是整数 1 到 n 的总和。你还能做得更好吗？将计算的值保存在内存中，这样在多次调用时是否就不必每次都计算呢？

4.4.1 命令式编程中的记忆化

在命令式编程中，你甚至不会有这个问题，因为处理方式显然如下所示：

```
public static void main(String args[]) {
    System.out.println(fibo(10));
}

public static String fibo(int limit) {
    switch(limit) {
        case 0:
            return "0";
        case 1:
            return "0, 1";
        case 2:
            return "0, 1, 1";
        default:
            BigInteger fibo1 = BigInteger.ONE;
            BigInteger fibo2 = BigInteger.ONE;
            BigInteger fibonacci;
            StringBuilder builder = new StringBuilder("0, 1, 1");
            for (int i = 3; i <= limit; i++) {
                fibonacci = fibo1.add(fibo2);
                builder.append(", ").append(fibonacci);
                fibo1 = fibo2;
                fibo2 = fibonacci;
            }
            return builder.toString();
    }
}
```

虽然这段程序集中了函数式编程中应该避免或解决的大多数问题，但是它能工作并且比你的函数式版本效率更高。原因就是记忆化。

记忆化是一种在内存中保留计算结果的技术，如果你需要在未来重做相同的计

算，它便可以立即返回。把记忆化应用于函数上，能够使函数记忆先前调用的结果，因此如果再次使用相同的参数调用，它们可以更快地返回结果。

这似乎与函数式的原则不兼容，因为记忆函数会维持状态。但并非如此，因为当调用的参数相同时，函数的结果也相同。（你甚至可以认为它不仅仅是相同，因为根本都不会再次计算！）存储结果的副作用不能在函数外部可见。

在命令式编程中，这甚至都没有被注意到。维持状态是计算结果的惯用手段，所以根本没有注意到记忆化。

4.4.2 递归函数的记忆化

递归函数经常隐式地使用记忆化。在斐波那契函数的递归示例中，你希望返回数列，因而计算了数列中的每个数字，导致不必要的重新计算。一个简单的解决办法是重写函数，以便直接返回表示数列的字符串。

练习 4.9

编写一个栈安全的尾递归函数，以整型 n 为参数，并返回一个字符串，表示斐波那契数列从 0 到 n 的值，用逗号加空格分隔。

提示

一个方案是使用 `StringBuilder` 作为累加器。`StringBuilder` 不是一个函数式结构，因为它是可变的，但这个变化并不会被外界可见。另一个方案是返回一个数字列表，然后将其转换为 `String`。这个方案更简单，因为你可以这样来抽象分隔符的问题：首先返回一个列表，然后编写一个函数将列表转换为一个逗号分隔的字符串。

答案 4.9

清单 4.5 展示了使用 `List` 作为累加器的方案。

清单 4.5 用隐式记忆化递归实现斐波那契数列

```
public static String fibo(int number) {  
    List<BigInteger> list = fibo_(list(BigInteger.ZERO),  
        BigInteger.ONE, BigInteger.ZERO, BigInteger.valueOf(number)).eval();  
    return makeString(list, ", ");  
}
```

调用 `fibo_` 辅助方法来获取斐波那契数字列表

```
private static TailCall<List<BigInteger>> fibo_(List<BigInteger> acc,
        BigInteger acc1, BigInteger acc2, BigInteger x) {
    return x.equals(BigInteger.ZERO)
        ? ret(acc)
        : x.equals(BigInteger.ONE)
        ? ret(append(acc, acc1.add(acc2)))
        : sus(() -> fibo_(append(acc, acc1.add(acc2)),
            acc2, acc1.add(acc2), x.subtract(BigInteger.ONE)));
}
```

```
public static <T> String makeString(List<T> list, String separator) {
    return list.isEmpty()
        ? ""
        : tail(list).isEmpty()
        ? head(list).toString()
        : head(list) + foldLeft(tail(list), "",
            x -> y -> x + separator + y);
}
```

调用 makeString 方法将列表
格式化为逗号分隔的字符串

递归还是反递归

本示例演示了使用隐式记忆化。不要认为这是解决问题的最佳方案。许多问题换一种方式更容易解决。所以让我们来改变一下思路。

你可以视斐波那契数列为一对对的数列（元组），而不是一组数字。不再这样生成，

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

你可以尝试这样生成：

(0, 1), (1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21), ...

在这个数列中，每个元组都可以由上一个元组构造出来。元组 n 的第二个元素变为元组 $n + 1$ 的第一个元素。元组 $n + 1$ 的第二个元素等于元组 n 的两个元素之和。在 Java 中，你可以这样编写一个函数：

```
x -> new Tuple<>(x._2, x._1.add(x._2));
```

现在可以用一个反递归方法代替递归方法：

```
public static String fiboCorecursive(int number) {
    Tuple<BigInteger, BigInteger> seed =
        new Tuple<>(BigInteger.ZERO, BigInteger.ONE);
    Function<Tuple<BigInteger, BigInteger>, Tuple<BigInteger, BigInteger>> f =
        x -> new Tuple<>(x._2, x._1.add(x._2));
```

```
List<BigInteger> list = map(List.iterate(seed, f, number + 1), x -> x._1);
return makeString(list, ", ");
}
```

`iterate` 方法接收一个种子 (`seed`)、一个函数，还有一个数字 n 为参数，并通过将函数应用于每个元素来计算下一个值的方式，来创建长度为 n 的列表。这是它的签名：

```
public static <B> List<B> iterate(B seed, Function<B, B> f, int n)
```

可以在 `fpinjava-common` 模块中找到这个方法。

4.4.3 自动记忆化

记忆化并不主要用于递归函数，它可以用来加速任何函数。想想你如何执行乘法。如果需要用 234 乘以 686，你可能需要一支笔和几张纸，或是一个计算器。但如果问 9 乘以 7，你便可以立即回答，而无须任何计算。这是因为你记住了乘法表。记忆函数的工作原理相同，它计算一次并保留结果。

假设你有一个函数式方法 `doubleValue`，它把参数乘以 2：

```
Integer doubleValue(Integer x) {
    return x * 2;
}
```

你可以通过将结果存储在一个 `map` 中来记忆该方法：

```
Map<Integer, Integer> cache = new ConcurrentHashMap<>(); // 用于存储结果
Integer doubleValue(Integer x) {
    if (cache.containsKey(x)) { // 查看 map 来确定是否结果已经被计算过了
        return cache.get(x); // 如果找到了，则返回结果
    } else {
        Integer result = x * 2; // 如果没找到，则计算结果
        cache.put(x, result); // 将结果放入
        return result; // 返回结果
    }
}
```

在 Java 8 里，它可以短很多：

```
Map<Integer, Integer> cache = new ConcurrentHashMap<>();

Integer doubleValue(Integer x) {
    return cache.computeIfAbsent(x, y -> y * 2);
}
```

如果你喜欢使用函数（考虑到本书的主题，可能性还是较大的），可以应用相同的原理：

```
Function<Integer, Integer> doubleValue =  
    x -> cache.computeIfAbsent(x, y -> y * 2);
```

但是出现了两个问题：

- 你必须重复修改所有要记忆的函数。
- 你用的 `map` 暴露给外界了。

第二个问题很容易解决。可以将方法或函数放在单独的类中，包括具有私有访问权限的 `map`。以下是一个方法的示例：

```
public class Doubler {  
  
    private static Map<Integer, Integer> cache = new ConcurrentHashMap<>();  
  
    public static Integer doubleValue(Integer x) {  
        return cache.computeIfAbsent(x, y -> y * 2);  
    }  
}
```

然后你可以实例化该类，并在每当要计算一个值时使用它：

```
Integer y = Doubler.doubleValue(x);
```

通过这种办法，`map` 不再能从外界访问了。你不能对函数照猫画虎，因为函数没有静态成员。一种办法是将 `map` 传递给函数作为附加参数。这可以通过闭包来完成：

```
class Doubler {  
    private static Map<Integer, Integer> cache = new ConcurrentHashMap<>();  
  
    public static Function<Integer, Integer> doubleValue =  
        x -> cache.computeIfAbsent(x, y -> y * 2);  
}
```

你可以如下使用该函数：

```
Integer y = Doubler.doubleValue.apply(x);
```

与方法的方案相比，这样做并没有什么优势。但是你也可以在更惯用的例子中

使用这个函数，例如：

```
map(range(1, 10), Doubler.doubleValue);
```

这相当于使用具有以下语法的方法版本：

```
map(range(1, 10), Doubler::doubleValue);
```

需求

你需要的是能实现以下代码的办法：

```
Function<Integer, Integer> f = x -> x * 2;
Function<Integer, Integer> g = Memoizer.memoize(f);
```

接下来你便可以使用记忆函数来代替原来的函数。就函数 *g* 返回的所有值而言，第一次都将通过原来的函数 *f* 计算，而以后的访问都将从缓存中返回。与此相反，如果你创建第三个函数，

```
Function<Integer, Integer> f = x -> x * 2;
Function<Integer, Integer> g = Memoizer.memoize(f);
Function<Integer, Integer> h = Memoizer.memoize(f);
```

由 *g* 缓存的值将不会被 *h* 返回；*g* 和 *h* 会使用单独的缓存。

实现

Memoizer 类很简单，如清单 4.6 所示。

清单 4.6 *Memoizer* 类

```
public class Memoizer<T, U> {

    private final Map<T, U> cache = new ConcurrentHashMap<>();

    private Memoizer() {}

    public static <T, U> Function<T, U> memoize(Function<T, U> function) {
        return new Memoizer<T, U>().doMemoize(function);
    }

    private Function<T, U> doMemoize(Function<T, U> function) {
        return input -> cache.computeIfAbsent(input, function::apply);
    }
}
```

记忆化方法返回其函数参数的记忆化版本

doMemoize 方法处理计算，并在必要时调用原来的函数

清单 4.7 展示了如何使用这个类。程序模拟长时间计算以展示记忆化函数的效果。

清单 4.7 Memoizer 示范

```
private static Integer longCalculation(Integer x) {
    try {
        Thread.sleep(1_000);
    } catch (InterruptedException ignored) {}
    return x * 2;
}

private static Function<Integer, Integer> f =
    MemoizerDemo::longCalculation;

private static Function<Integer, Integer> g = Memoizer.memoize(f);

public static void automaticMemoizationExample() {
    long startTime = System.currentTimeMillis();
    Integer result1 = g.apply(1);
    long time1 = System.currentTimeMillis() - startTime;
    startTime = System.currentTimeMillis();
    Integer result2 = g.apply(1);
    long time2 = System.currentTimeMillis() - startTime;
    System.out.println(result1);
    System.out.println(result2);
    System.out.println(time1);
    System.out.println(time2);
}
```

模拟长时间计算

需要被记忆的函数

已经被记忆的函数

在我的电脑上运行 `automaticMemoizationExample` 方法会生成以下结果：

```
2
2
1000
0
```

请注意，精确的结果将取决于电脑的速度。

你现在只需调用一个方法，便能够使函数成为记忆化函数而变得不再平凡。但是要在生产环境中使用这种技术，你还需要处理潜在的内存问题。如果可能的输入数量较少，则此代码还可以接受，因此可以将所有的结果保存在内存中，而不会导致内存溢出。否则，你可以使用软引用（soft reference）或者弱引用（weak reference）来存储记忆值。

“多参”函数的记忆化

正如我先前所说，在这个世界上并没有函数接收多个参数。函数是一个集（源集）

到另一个集（目标集）的应用，它们不能有多参数。看起来有多参数的函数是以下两者之一：

- 元组函数。
- 函数返回函数返回函数……返回一个结果。

无论是哪一种情况，你只关心单参函数，所以可以轻易使用你的 Memoizer 类。

使用元组的函数可能是最简单的选择。你可以使用在前面章节里编写的 Tuple 类，但是要在 map 中存储元组，你必须实现 equals 和 hashCode 方法。除此之外，你还需要定义两个元素（一对）的元组、三个元素的元组，等等。谁知道该在哪里停下来？

第二个选项更容易。需要使用柯里化版本的函数，就像在前面的章节里做过的。记忆柯里化函数很容易，虽然不能和上面一样简单。需要记忆每一个函数：

```
Function<Integer, Function<Integer, Integer>> mhc =
    Memoizer.memoize(x ->
        Memoizer.memoize(y -> x + y));
```

可以使用相同的技术来记忆三个参数的函数：

```
Function<Integer, Function<Integer, Function<Integer, Integer>>> f3 =
    x -> y -> z -> x + y - z;

Function<Integer, Function<Integer, Function<Integer, Integer>>> f3m =
    Memoizer.memoize(x ->
        Memoizer.memoize(y ->
            Memoizer.memoize(z -> x + y - z)));
```

清单 4.8 展示了使用具有三个参数的记忆函数的示例。

清单 4.8 测试具有三个参数的记忆函数的性能

```
Function<Integer, Function<Integer, Function<Integer, Integer>>> f3m =
    Memoizer.memoize(x ->
        Memoizer.memoize(y ->
            Memoizer.memoize(z ->
                longCalculation(x) + longCalculation(y) - longCalculation(z))));

public void automaticMemoizationExample2() {
    long startTime = System.currentTimeMillis();
    Integer result1 = f3m.apply(2).apply(3).apply(4);
    long time1 = System.currentTimeMillis() - startTime;
    startTime = System.currentTimeMillis();
    Integer result2 = f3m.apply(2).apply(3).apply(4);
```

```

long time2 = System.currentTimeMillis() - startTime;
System.out.println(result1);
System.out.println(result2);
System.out.println(time1);
System.out.println(time2);
}

```

这段程序生成了以下输出：

```

2
2
3002
0

```

这表明对 longCalculation 方法的第一次访问需要 3000 毫秒，而第二次访问立即返回。

另一方面，在定义了 Tuple 类之后，使用元组的函数看起来更加容易。清单 4.9 展示了 Tuple3 的示例。

清单 4.9 Tuple3 的一个实现

```

public class Tuple3<T, U, V> {

    public final T _1;
    public final U _2;
    public final V _3;

    public Tuple3(T t, U u, V v) {
        _1 = Objects.requireNonNull(t);
        _2 = Objects.requireNonNull(u);
        _3 = Objects.requireNonNull(v);
    }

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Tuple3)) return false;
        else {
            Tuple3 that = (Tuple3) o;
            return _1.equals(that._1) && _2.equals(that._2)
                && _3.equals(that._3);
        }
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + _1.hashCode();
    }
}

```



```

    result = prime * result + _2.hashCode();
    result = prime * result + _3.hashCode();
    return result;
}
}

```

清单 4.10 展示了测试以 Tuple3 为参数的记忆函数的一个示例。

清单 4.10 Tuple3 的记忆函数

```

Function<Tuple3<Integer, Integer, Integer>, Integer> ft =
    x -> longCalculation(x._1)
        + longCalculation(x._2)
        - longCalculation(x._3);
Function<Tuple3<Integer, Integer, Integer>, Integer> ftm =
    Memoizer.memoize(ft);

public void automaticMemoizationExample3() {
    long startTime = System.currentTimeMillis();
    Integer result1 = ftm.apply(new Tuple3<>(2, 3, 4));
    long time1 = System.currentTimeMillis() - startTime;
    startTime = System.currentTimeMillis();
    Integer result2 = ftm.apply(new Tuple3<>(2, 3, 4));
    long time2 = System.currentTimeMillis() - startTime;
    System.out.println(result1);
    System.out.println(result2);
    System.out.println(time1);
    System.out.println(time2);
}

```

记忆函数是纯函数吗

记忆即关于保持函数调用之间的状态。记忆函数是行为取决于当前状态的函数，但它总是会为相同的参数返回相同的值，只是返回值所需的时间会有所不同。因此只要原来的函数是纯函数，记忆函数仍然是一个纯函数。

时间的变化可能是一个问题。像原来的斐波那契函数那样需要很多年才能完成的函数，也许可以称为永不止步，因而时间的增加可能会产生问题。另一方面，让函数变得更快不应该是一个问题。否则，其他地方就会有一个更大的问题！

4.5 总结

- 递归函数是一个定义为引用自己的函数。
- 在 Java 中，递归方法在递归调用自己之前将当前计算状态推入栈。

- Java 默认的栈的大小有限。它可以配置得更大，但通常只是浪费空间，因为所有线程都使用相同的栈大小。
- 尾递归函数是递归调用发生在最后（尾部）位置的函数。
- 在某些语言中，递归函数使用尾调用消除（TCE）来优化。
- Java 没有实现 TCE，但是可以模拟它。
- lambda 可以递归。
- 记忆化可以让函数记忆它们的计算结果，以便加速之后的访问。
- 可以实现自动记忆化。

5 用列表处理数据

本章要点

- 在函数式编程中对数据结构进行分类
- 使用常见的单链表
- 理解不可变性的重要性
- 用递归和函数处理列表

数据结构是编程中乃至生活中最重要的概念。我们看到的世界本身就是一个巨大的数据结构，由简单一些的数据结构组成，而这些简单的数据结构又是由更加简单的结构所组成。每当我们想要对什么东西建模时，无论是对象还是现实，我们最终都会得到数据结构。

数据结构有许多种。在计算机领域，数据结构经常通过术语集合（collection）表示一个整体。一个集合是一组相互有着某些关系的数据项。最简单的关系就是它们事实上都属于相同的组。

5.1 如何对数据集合进行分类

可以从很多种角度对数据集合进行分类。你可以用线性（linear）、关联

(associative) 和图 (graph) 来分类：

- 线性集合是元素被单向关联的集合。在线性集合里，每个元素都维持着与下一个元素的关联。列表 (list) 是最常见的线性集合示例。
- 关联集合是可以作为函数查看的集合。给定对象 o ，一个 $f(o)$ 函数将会按照对象是否从属于集合而返回 `true` 或 `false`。与线性集合不同，关联集合内部的元素没有任何关联。尽管可以定义元素的顺序，但这些集合事实上是无序的。集 (set) 和关联数组 (也被称为 `map` 或者 `dictionary`) 是关联集合最常见的示例。我们将会在第 11 章中学习 `map` 的函数式实现。
- 图是元素与多个其他元素相关联的集合。树是一个特殊的示例，更加特定的是二叉树 (binary tree)，每个元素都与另外两个元素相关联。你将在第 10 章中学习树的函数式观点。

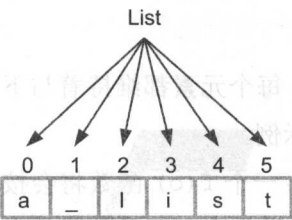
5.1.1 不同的列表类型

在本章中，我们将会关注线性集合中最常见的类型，列表。列表是在函数式编程中最常用的数据结构，所以经常用于函数式编程概念的教学。然而你需要知道的是，本章中学的这些并不仅限于列表，还适用于许多其他数据结构（不一定是集合）。

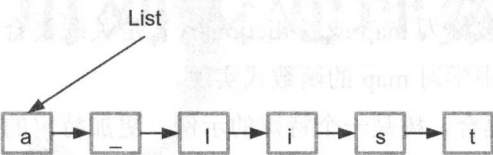
列表可以从如下几个不同方面来进一步分类。

- 访问——一些列表只能从一端访问，其他的可以从两端访问。还有一些从一端写入，从另一端读取。最后，有些列表允许通过位置来访问任意元素，元素的位置也称为索引 (index)。
- 排序类型——一些列表的元素是按照它们被插入的顺序读取的。这种结构类型叫作先入先出 (FIFO, first in, first out)。还有一种与插入的顺序相反 (LIFO, last in, first out)。最后，有些列表允许用截然不同的顺序获取元素。
- 实现——访问类型和顺序是与你选择的列表实现强相关的概念。如果选择用链表类的列表而不是索引数组类的列表实现，从访问的角度上，你就会得到一个完全不同的结果。如果选择的是双向链表，那你就得到一个从两端都可以访问的列表。

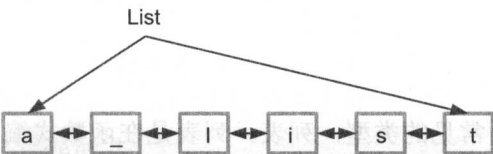
图 5.1 展示了不同的列表提供了不同的访问方式。请注意这张图展示了隐藏在每种列表类型背后的原则，但并非实现列表的方式。



通过索引列表，可以直接访问任意元素。



单向链表只允许访问其第一个元素。



双向链表允许从两端访问。

图 5.1 不同的列表类型提供了不同的元素访问方式。

5.1.2 对列表性能的相对期望

对各种操作的性能期望是选择列表类型的一个重要标准。性能一般用大写的 O 来表示。这个符号主要用于数学，但是当用于计算机时，它指的是当输入大小变化时的算法复杂度。当用于描述列表操作性能的时候，这个符号显示了随着列表长度的变化，函数的性能将如何变化。例如，思考以下性能：

- $O(1)$ ——这意味着操作所需的时间是一个常量（你可以理解为： n 个元素的耗时相当于一个元素的耗时乘以 1）。
- $O(\log(n))$ ——这意味着 n 个元素的耗时相当于一个元素的耗时乘以 $\log(n)$ 。
- $O(n)$ ——这意味着 n 个元素的耗时相当于一个元素的耗时乘以 n 。
- $O(n^2)$ ——这意味着 n 个元素的耗时相当于一个元素的耗时乘以 n^2 。

如果数据结构对所有类型的操作都是 $O(1)$ 的性能，那就再理想不过了。可惜的是，迄今为止这还是不可能的。每种列表类型对不同的操作都有着不同的性能。索引列表对获取数据可以提供 $O(1)$ 的性能，对插入数据为近乎 $O(1)$ 。单向链表可以

对插入数据和获取一端的数据提供 $O(1)$ 的性能,但是如果需要获取的数据在另一端,则是 $O(n)$ 。

选择最佳的结构是一种取舍。大多数时候,对大部分频繁的操作而言,你需要 $O(1)$ 的性能,并且不得不接受非常用操作的 $O(\log(n))$ 甚至是 $O(n)$ 的性能。

注意,这种测量性能的方法对结构有一个要求,要求它可以无限扩展。这不适用于我们能控制的数据结构,因为你的结构受限于可用的内存大小。大小的限制可能会导致某个结构的 $O(n)$ 操作比另一个结构的 $O(1)$ 还快。如果第一个结构的单个元素操作时间极短,内存的限制可能会让第二个结构无法体现出自己的优势。操作时间为 1 纳秒的 $O(n)$ 性能经常比 1 毫秒的 $O(1)$ 要好。(后者只有当列表超过 1 000 000 个元素时才会比前者更快。)

5.1.3 时间与空间,时间与复杂度的取舍

你刚刚看到了选择数据结构的实现一般是时间与空间的取舍。视哪种操作更频繁而定,你会选择一个某些操作更快,而其他操作更慢的实现。不过还有其他需要取舍的地方。

想象一下,你需要从一个结构里以指定的顺序来获取元素,最小的最先被取出。你可以选择在插入时排序元素,也可以选择插入时仅存储,而只在获取时检索最小值。做出决定的一个重要标准是:获取到的数据是否从结构中移除。如果不是,那它可能会在没有被删除的情况下被访问许多次,更好的选择可能是在插入时排序元素以免在获取时多次排序。这个用例与优先队列 (priority queue) 相对应,你用它等待一个给定的元素。你可以多次测试这个队列直到返回期望的元素。这样的用例需要在插入元素时进行排序。

但是如果你想用几个不同的顺序来访问元素呢?例如,你可能想要以插入顺序或是反序来访问元素。这个结果与图 5.1 所示的双向链表相对应。看起来在这样的情况下,元素应该在被获取时排序。你可能会青睐于单向链表,可以在一端达到 $O(1)$ 的访问时间而在另一端为 $O(n)$ 。或者发明一个不同的结构,让两端都能达到 $O(\log(n))$ 。另一个方案是存储两个列表,一个是插入顺序而另一个是反序。这样的话,在插入时会慢一些,但是从两端访问都是 $O(1)$ 。这个方案的一个缺点是会消耗更多的内存。因此你能发现选择正确的结构也是一个在时间和空间中取舍的问题。

但是你也可能发明出一些从两端插入和获取元素都可以做到最优的结构。这些

结构类型都被发明出来了，你只需实现它们即可。不过这样的结构比最简单的结构要复杂得多，所以也是在时间与复杂度之间进行取舍。

5.1.4 直接修改

大部分数据结构由于元素的插入和删除而经常变化。基本上有两种办法可以处理这样的操作。第一种是直接修改。

直接修改包括通过修改数据结构自身来改变结构里的元素。如果所有的程序都是单线程的话，这是一个好主意，然而并非如此。当所有的程序都是多线程时麻烦就大了。这样做不仅仅需要考虑元素替换，还有增加、删除、排序和其他所有改变结构的操作。如果程序允许改变数据结构，在没有严密保护这些结构时它们无法共享，而保护的难度极大以至于一般第一次实现都做得不对，从而陷入死锁、活锁、线程饥饿、脏数据等诸如此类的麻烦事。

那么解决方案是什么呢？使用不可变的数据结构就行了。许多命令式程序员在第一次知道这一点的时候都震惊了。如果不能改变数据结构，如何才能用它们来做点有用的事？毕竟，你经常始于空的结构并且想要往里面增加数据。如果它们不可变，那么应该怎么做？

直接修改

在一篇 1981 年的文章《事务的概念：优势和限制》里，Jim Gray 写道：¹

直接修改：有毒的苹果？

当使用泥简或纸墨记账（bookkeeping）时，会计师就良好的会计实践制订了一些明确的规则。其中一个基本规则是复式记账法（double-entry bookkeeping），计算相当于自检，从而尽快发现错误记录。第二条规则是决不修改账簿。如果发生错误，则为其添加注释并在账簿中增加一条新的补偿记录。因此，这些账簿是商业交易的完整历史……

¹ Jim Gray, 《事务的概念：优势和限制》（Tandem 计算机，技术报告 81.3，1981 年 6 月），<http://www.hpl.hp.com/techreports/tandem/TR-81.3.pdf>。

许多系统设计师视直接修改为一个原罪：它违背了已经遵守了数百年的传统会计实践。

答案很简单。正如使用复式记账法来代替以前的直接修改法，你用新建的数据来代表新的状态。创建一个包含新元素的新列表，而不是从一个现有列表里增加新元素。最主要的好处就是如果另一个线程在插入时也操作了这个列表，它就不会受到变更的影响，因为它根本就看不到新列表。

一般来说，这种说法立即会引发两大抗议：

- 如果另一个线程感知不到变更，那它就是在操作过期的旧数据。
- 创建一个列表的全新副本既消耗时间又消耗内存，从而导致不可变的数据结构性能很差。

这两个论点都靠不住。实际上操作“旧数据”的线程使用的数据正是它一开始所读取的数据。如果插入元素发生在另一线程操作结束之后，那就没有并发问题。但是如果插入发生在操作正在进行的时候，对于可变数据结构来说会发生什么事呢？要么是对并发操作没有保护，数据被破坏或是结果不正确（或者二者兼有），要么就是保护机制锁住了数据，将插入操作延迟至第一个线程结束之后。对于后者来说，结果与不可变结构完全相同。

5.1.5 持久化数据结构

正如你在上一节所见，在插入元素之前生成一个数据结构的副本（有时称为防御性副本）经常被认为是一个低性能的耗时操作。如果你用了数据共享确实不会这样，因为不可变数据结构是持久化的，所以确实有可能。图 5.2 展示了如何能够在最佳性能下将元素移除和添加到一个新的、不可变的单向列表中。

如你所见，完全没有任何副本。结论就是，这样的列表与可变列表相比，对于删除和插入元素而言可能会有更佳的性能。所以函数式数据结构（不可变和持久化）并不总是比可变版本慢，它们甚至经常更快（虽然在有些操作上可能更慢）。不管在什么场合，它们总是更安全的。

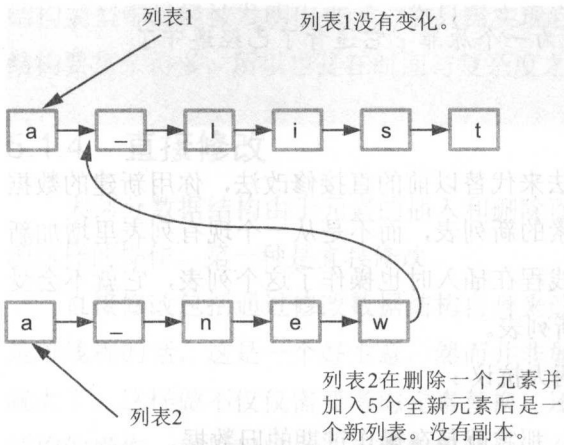


图 5.2 删除和增加元素时不做修改或使用副本。

5.2 一个不可变、持久化的单链表实现

图 5.1 和图 5.2 里展示的单链表结构只是理论。列表不能那样实现，因为元素无法一个个地被链接起来，除非它们是允许链接的特殊元素，可你想要的列表是能够存储任何元素的。解决方案是设计一个由以下元素组成的递归列表结构：

- 排在列表第一位的元素，也称为 *head*。
- 列表的剩余部分还是一个列表，称为 *tail*。

请注意，你已经遇到了一个由两种不同类型的元素所组成的泛型元素：Tuple。一个元素为 A 类型的单向链表其实就是一个 Tuple<A, List<A>>。接下来你就可以定义列表为

```
class List<A> extends Tuple<A, List<A>>
```

但是正如我在第 4 章中解释的那样，你需要一个终止情况，就像在每个递归定义里所做的那样。根据约定，这个终止情况被称为 Nil，并与空列表相对应。由于 Nil 并没有 head，也没有 tail，因此它并不是一个 Tuple。你新定义的列表就是以下二者之一：

- 一个空列表 (Nil)
- 一个由元素和列表组成的元组

你将创建一个专门持有 head 和 tail 这两个属性的 List 类，而不是直接使用属性为 _1 和 _2 的 Tuple。这样可以简化处理 Nil 的情况。图 5.3 展示了列表实现的结构：

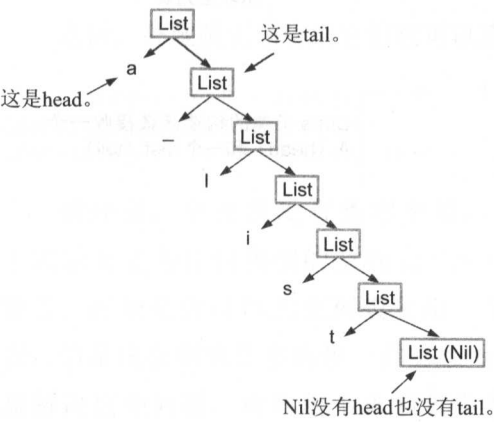


图 5.3 单链表实现的表现形式。

清单 5.1 展示了这个列表的基本实现。

清单 5.1 单链表

```
public abstract class List<A> {  
    public abstract A head();  
    public abstract List<A> tail();  
    public abstract boolean isEmpty();  
    @SuppressWarnings("rawtypes")  
    public static final List NIL = new Nil();  
    private List() {}  
    private static class Nil<A> extends List<A> {  
        private Nil() {}  
        public A head() {  
            throw new IllegalStateException("head called en empty list");  
        }  
        public List<A> tail() {  
            throw new IllegalStateException("tail called en empty list");  
        }  
        public boolean isEmpty() {  
            return true;  
        }  
    }  
}
```

Annotations:

- List 实现为一个抽象类，类型参数为自己元素的类型，由类型变量 A 表示
- 用一个单例来表示空列表
- Nil（不在列表里）子类表示空列表
- Nil 子类的无参私有构造函数

```

    }
}

private static class Cons<A> extends List<A> {

    private final A head;
    private final List<A> tail;

    private Cons(A head, List<A> tail) {
        this.head = head;
        this.tail = tail;
    }

    public A head() {
        return head;
    }

    public List<A> tail() {
        return tail;
    }

    public boolean isEmpty() {
        return false;
    }
}

@SuppressWarnings("unchecked")
public static <A> List<A> list() {
    return NIL;
}

@SafeVarargs
public static <A> List<A> list(A... a) {
    List<A> n = list();
    for (int i = a.length - 1; i >= 0; i--) {
        n = new Cons<>(a[i], n);
    }
    return n;
}

```

Cons (construct) 子类表示非空列表

Cons 子类的构造函数接收一个 A (head) 和一个 List (tail)

一个用于构造空列表的静态工厂方法

一个用于构造非空列表的静态工厂方法

由于需要首先插入最后一个元素，所以反向处理索引。从访问性的角度上说，单向链表其实就是栈

List 类实现为一个抽象类。它包含两个私有的静态子类，用于表示 List 的两个可能：空列表的 Nil 和非空列表的 Cons。

List 类定义了三个抽象方法：head()，用于返回列表的第一个元素；tail()，用于返回列表的剩余元素（除了第一个元素以外）；还有 isEmpty()，用于当列表为空时返回 true，否则返回 false。List 类的类型参数为 A，表示列表内的元素类型。

子类是私有的，这样你就只能通过调用静态工厂方法来创建列表了。可以静态导入这些方法：

```
import static fpinjava.datastructures.List.*;
```

之后，外围类无须引用它们也可以直接使用，如下所示：

```
List<Integer> ex1 = list();  
List<Integer> ex2 = list(1);  
List<Integer> ex3 = list(1, 2);
```

请注意，空列表没有类型参数。换句话说，它是一个原始类型，可以用于表示元素为任何类型的空列表。因此，编译器会在创建或使用空列表时发出警告。好处是你可以为空列表使用一个单例。另一个办法是使用一个泛型空列表，但是这会招致许多麻烦。你必须为每个类型参数创建一个不同的空列表。要想解决这个问题，请使用不含参数类型的空列表单例。这样做编译器会发出警告。为了将这个警告限制在 `List` 类中，而不至于将其暴露给 `List` 的用户，你不能直接访问单例。这就是为什么会有一个访问单例的（泛型）静态方法，在 `NIL` 属性上的 `@SuppressWarnings("rawtypes")`，还有在 `list()` 方法上的 `@SuppressWarnings("unchecked")`。

请注意，`list(A ... a)` 方法使用了 `@SafeVarargs` 注解，表示该方法不会导致堆污染（heap pollution）。这个方法使用了基于 `for` 循环的命令式实现。这不是很“函数式”，但它是在难度和性能上的一个折中。如果你坚持以函数式的方式去实现它，当然没问题。你所需要的全部，就是一个接收数组为参数的函数，并返回其最后一个元素，以及另一个返回除了最后一个元素以外的数组的函数。以下是一个可行的方案：

```
@SafeVarargs  
public static <A> List<A> list(A... as) {  
    return list_(list(), as).eval();  
}  
  
public static <A> TailCall<List<A>> list_(List<A> acc, A[] as) {  
    return as.length == 0  
        ? ret(acc)  
        : sus(() -> list_(new Cons<>(as[as.length - 1], acc),  
            Arrays.copyOfRange(as, 0, as.length - 1)));
```

但是，请确保不要使用这个实现，因为它比命令式的实现慢 10 000 倍。这就是

不要盲目地追求函数式的一个绝佳示例。命令式版本有一个函数式接口，而这就是你所需要的。请注意递归并不是问题，使用 TailCall 的递归几乎和迭代一样快。这里的问题是 copyOfRange 方法，它真的是非常慢。

5.3 在列表操作中共享数据

像单链表这样的不可变持久化数据结构的一大优点就是数据共享所能提供的性能提升。你已经看到了访问列表的第一个元素是即时的。只需调用一下 head() 方法，而 head() 方法是 head 属性的一个简单的访问器。

删除第一个元素也一样快。只需调用 tail() 方法，它将返回 tail 属性。现在让我们看看如何添加一个元素来获得一个新的列表。

练习 5.1

实现函数式实例方法 cons，在列表的开头添加一个元素。（记住，cons 代表 construct。）

答案 5.1

这个实例方法在 Nil 和 Cons 子类上的实现相同：

```
public List<A> cons(A a) {  
    return new Cons<>(a, this);  
}
```

练习 5.2

实现实例方法 setHead，用新值替换 List 的第一个元素。

答案 5.2

你可能会想为此实现一个静态方法，但是需要测试是否为空列表：

```
public static <A> List<A> setHead(List<A> list, A h) {  
    if (list.isEmpty()) {  
        throw new IllegalStateException("setHead called on an empty list");  
    } else {  
        return new Cons<>(h, list.tail());  
    }  
}
```

然而这并没有什么意义。作为一个通用规则，如果你发现自己被迫使用 if...

else 结构，那可能这条路走错了。思考如何实现调用这个静态方法的实例方法。

一个更佳方案是往 List 类中添加一个抽象方法：

```
public abstract List<A> setHead(A h);
```

Nil 子类的实现很直观。抛出异常就好了，因为试图访问一个空列表的 head 就是一个 bug：

```
public List<A> setHead(A h) {
    throw new IllegalStateException("setHead called on empty list");
}
```

Cons 实现对应于静态方法的 else 分支：

```
public List<A> setHead(A h) {
    return new Cons<>(h, tail());
}
```

如果你需要一个静态方法，它可以简单地调用实例方法的实现：

```
public static <A> List<A> setHead(List<A> list, A h) {
    return list.setHead(h);
}
```

练习 5.3

编写一个 toString 方法以显示列表的内容。空列表将显示为 "[NIL]"，包含从 1 到 3 的整型列表将显示为 "[1, 2, 3, NIL]"。对于任意对象的列表，将调用每个对象的 toString 方法来显示。

答案 5.3

Nil 的实现非常简单：

```
public String toString() {
    return "[NIL]";
}
```

Cons 是一个递归实现，使用 StringBuilder 作为累加器。请注意，虽然 StringBuilder 是一个可变对象，但它有一个对函数式友好的 append 方法，因为它返回更改后的 StringBuilder 实例。

```
public String toString() {
    return String.format("[%sNIL]",
```

```

    toString(new StringBuilder(), this).eval());
}

private TailCall<StringBuilder> toString(StringBuilder acc, List<A> list) {
    return list.isEmpty()
        ? ret(acc)
        : sus(() -> toString(acc.append(list.head()).append(", "),
                               list.tail()));
}

```

如果你想不起来如何使 TailCall 类在堆上而非在栈上递归，请参考第 4 章。

5.3.1 更多列表操作

依靠数据共享，你能够以非常高效的方式实现各种其他操作——通常比可变量列表更高效。在本节的剩余部分中，你将基于数据共享为链表添加功能。

练习 5.4

虽然 tail 方法不会以任何方式改变列表，但其效果与删除第一个元素相同。编写一个更通用的 drop 方法，从列表中删除前 n 个元素。当然，这个方法不会删除元素，但是会返回一个与预期结果相对应的新列表。这个“新”列表并没有什么新东西，因为将使用数据共享，所以不会创建任何东西。图 5.4 展示了应该如何继续。

方法的签名是

```
public List<A> drop(int n);
```

提示

你应该使用递归来实现 drop 方法。不要忘记考虑各种特殊情况，例如空列表或 n 大于列表长度等。

答案 5.4

你可以选择实现一个静态方法或实例方法。如果打算使用更易阅读的对象标记，那就必须实现实例方法。假如要删除整数列表的两个元素，然后将结果的第一个元素替换为 0，则可以使用静态方法：

```
List<Integer> newList = setHead(drop(list, 2), 0);
```

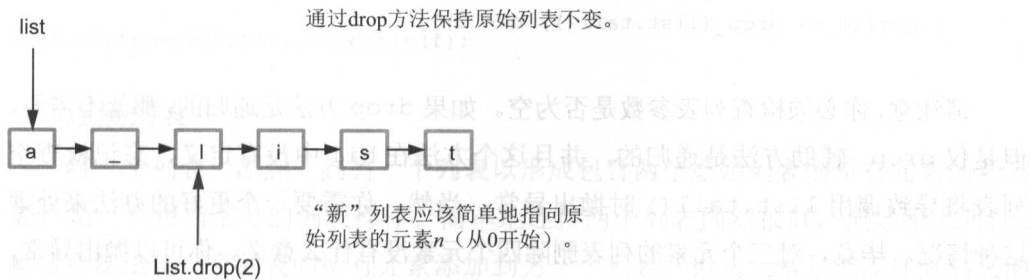


图 5.4 删除列表的前 n 个元素，而不改变或创建任何东西。

每当增加一个方法时，方法名称将被添加到左侧，包括列表本身的其他参数将被添加到右侧，如图 5.5 所示。



图 5.5 没有对象标记可能导致复合函数难以阅读。使用对象标记可以收获更多易读的代码。

使用对象标记使代码更易阅读：

```
List<Integer> newList = drop(list, 2).setHead(0);
```

drop 方法在 Nil 类中的实现就是简单地返回 this：

```
public List<A> drop(int n) {
    return this;
}
```

在 Cons 类中，使用私有辅助方法来实现递归，这与第 4 章中所学相同。这段代码假设静态导入了 TailCall.ret 和 TailCall.sus 方法：

```
public List<A> drop(int n) {
    return n <= 0
        ? this
        : drop_(this, n).eval();
}

private TailCall<List<A>> drop_(List<A> list, int n) {
    return n <= 0 || list.isEmpty()
        ? ret(list)
```



```

        : sus() -> drop_(list.tail(), n - 1));
    }

```

请注意，你必须检查列表参数是否为空。如果 `drop` 方法是递归的，那就不需要。但是仅 `drop_` 辅助方法是递归的，并且这个方法在 `Nil` 中没有定义，忘记检查空列表将导致调用 `list.tail()` 时抛出异常。当然，你需要一个更好的办法来处理这种情况。毕竟，对三个元素的列表删除四个元素没有什么意义。你可以抛出异常，但是最好使用将在下一章中学到的更加函数式的技术。

练习 5.5

实现 `dropWhile` 方法，只要条件为真，就删除 `List` 的 `head` 元素。这是添加到 `List` 抽象类里的签名：

```
public abstract List<A> dropWhile(Function<A, Boolean> f);
```

答案 5.5

我们不再关注 `Nil` 的实现，因为它只会返回 `this`。`Cons` 类是递归实现：

```

@Override
public List<A> dropWhile(Function<A, Boolean> f) {
    return dropWhile_(this, f).eval();
}

private TailCall<List<A>> dropWhile_(List<A> list,
                                     Function<A, Boolean> f) {
    return !list.isEmpty() && f.apply(list.head())
        ? sus() -> dropWhile_(list.tail(), f)
        : ret(list);
}

```

请注意，当在空列表上调用 `dropWhile` 时，可能会遇到问题。例如，以下代码不能通过编译：

```
list().dropWhile(f)
```

原因是 `Java` 无法从传递给 `dropWhile` 方法的函数中推断列表的类型。假设你正在处理一个整型列表，可以使用这个方案：

```

List<Integer> list = list();
list.dropWhile(f);

```

或者这样：

```
List.<Integer>list().dropWhile(f);
```

连接列表

将一个列表“添加”到另一个列表以形成包含两个原始列表的所有元素的新列表，是一个相当常见的列表操作。简单地连接两个列表固然很好，但这不太可能。解决办法是将一个列表的所有元素添加到另一个列表。但是元素只能添加到列表的前面（head），因此，如果要将 list1 连接到 list2 上，首先必须将 list1 的最后一个元素添加到 list2 的前面，如图 5.6 所示。

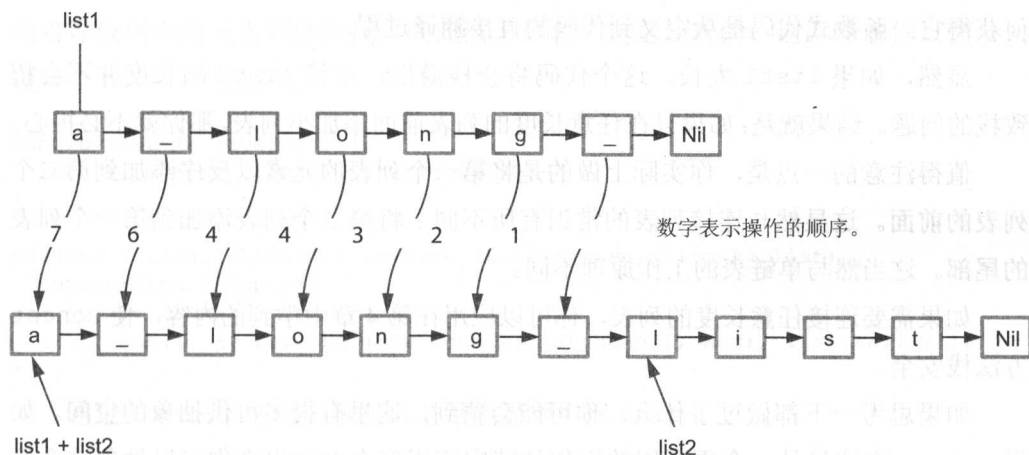


图 5.6 通过连接共享数据。你可以看到两个列表都保留下来了，并且结果列表共享了 list2。但是你也可以看到不能完全按照图中所示的进行，因为必须首先访问 list1 的最后一个元素，由于列表的结构，这是不可能的。

一种办法是首先反转 list1，生成一个新列表，然后将每个元素添加到 list2 中，这回从反向列表的 head 开始。但是你还没有定义反转列表的方法。那还能定义 concat 吗？可以。你只需考虑如何定义这个方法：

- 如果 list1 为空，则返回 list2。
- 否则返回 list1 的第一个元素（list1.head）加上 list1 的剩余部分（list1.tail）与 list2 的连接。

这个递归定义可以转换为以下代码：

```
public static List<A> concat(List<A> list1, List<A> list2) {  
    return list1.isEmpty()
```

```
? list2
: new Cons<>(list1.head(), concat(list1.tail(), list2));
}
```

这个解决方案的优点（对于一些读者来说）是，你不需要一张图来揭示它的工作原理，因为它没有“工作”，它只是一个翻译成代码的数学定义。

这个定义的主要缺点（对于其他读者来说）是，由于同样的原因，你很难轻易地在一张图中表示它。听起来像个小幽默，但并非如此。两个解决方案表示完全相同的“操作”，但是一个表示过程（从中可以看到结果），而另一个直接表示结果。哪一个更好是一个选择的问题。但是函数式编程通常要的是结果是什么，而不是如何获得它。函数式代码是从定义到代码的直接翻译过程。

显然，如果 `list1` 太长，这个代码将会栈溢出，尽管 `list2` 的长度并不会招致栈的问题。结果就是：如果只在任意长度的列表前面添加小列表，那就不必担心。

值得注意的一点是，你实际上做的是将第一个列表的元素以反序添加到第二个列表的前面。这显然与连接列表的常识有所不同：将第二个列表添加到第一个列表的尾部。这当然与单链表的工作原理不同。

如果需要连接任意长度的列表，你可以应用在第4章中学到的内容，使 `concat` 方法栈安全。

如果思考一下都做过了什么，你可能会猜到，这里有很多可供抽象的空间。如果 `concat` 方法只是一个更通用的操作的特定应用怎么办？也许你可以抽象这个操作，使它栈安全，然后重新用它来实现许多其他操作？等着瞧！

你可能已经注意到，这个操作的复杂性（以及 Java 的执行时间）与第一个列表的长度成比例。换句话说，如果连接长度为 n_1 和 n_2 的 `list1` 和 `list2`，复杂度为 $O(n_1)$ ，这意味着它与 n_2 无关。换句话说，视 n_2 而定，这个操作说不定会比在命令式 Java 中连接两个可变列表效率更高。

从列表末尾删除

有时需要从列表末尾删除元素。虽然单链表并不是这种操作的理想数据结构，但你仍然需要能够实现它。

练习 5.6

编写一个方法从列表中删除最后一个元素。该方法返回的结果应该是删除之后的列表。将其实现为实例方法，签名如下：

```
List<A> init()
```

提示

我们已经谈到过一个函数，也许可以用它来表示这个函数。也许现在正是创建这个辅助函数的时机。

答案 5.6

为了删除最后一个元素，你必须遍历列表（从前到后）并构建新列表（从后到前，因为列表中的“最后一个”元素必须为 Nil）。这是使用 Cons 对象来创建列表的结果。这将导致列表的元素顺序相反，因此结果列表一定是反转的。这意味着你只需要实现一个 reverse 方法即可：

```
public List<A> reverse() {
    return reverse_(list(), this).eval();
}

private TailCall<List<A>> reverse_(List<A> acc, List<A> list) {
    return list.isEmpty()
        ? ret(acc)
        : sus(() -> reverse_(new Cons<>(list.head(), acc), list.tail()));
}
```

通过 reverse 方法，你可以非常容易地实现 init：

```
public List<A> init() {
    return reverse().tail().reverse();
}
```

当然，这些是 Cons 类的实现。在 Nil 类中，reverse 方法返回 this，init 方法抛出一个异常。

5.4 使用高阶函数递归折叠列表

在第 3 章中，你学习了如何折叠列表，折叠也适用于不可变列表。但是对于可变列表而言，你可以选择通过迭代或递归来实现这些操作。在第 3 章中，你迭代地实现了折叠，因为你用的是通过非函数式方法添加和删除元素的可变列表。add 方法什么也不返回，而 remove 方法仅返回已删除的元素，同时还修改了列表参数。因为不可变列表是递归的数据结构，所以可以很容易地使用递归来实现折叠操作。

让我们思考在数字列表上通用的折叠操作。

练习 5.7

编写一个函数式方法，使用简单的基于栈的递归来计算整型列表内所有元素的总和。

答案 5.7

列表所有元素的总和的递归定义是

- 对于空列表：0。
- 对于非空列表：head 加上 tail 的总和。

这个定义几乎可以逐字逐句地翻译为 Java 代码：

```
public static Integer sum(List<Integer> ints) {  
    return ints.isEmpty()  
        ? 0  
        : ints.head() + sum(ints.tail());  
}
```

别忘了这个实现会在操作长列表时造成栈溢出，所以不要在生产环境中使用这类代码。

练习 5.8

编写一个函数式方法，使用基于栈的递归来计算双精度列表内所有元素的乘积。

答案 5.8

非空列表内所有元素的乘积的递归定义是

head * tail 的乘积

但是，空列表应该返回什么呢？当然，如果还能回想起数学课上的内容，你会知道答案。否则，你可以在答案 5.7 中展示的非空列表的需求中找到答案。

思考将递归公式应用到所有元素时会发生什么。你最终得到的结果必须乘以一个空列表内所有元素的乘积。因为最终想要得到这个结果，你只能认为一个空列表内所有元素的乘积为 1，别无选择。这与求和示例的情况相同，那时你把 0 当作空列表内所有元素的总和。求和操作的单位元，或者说是中性元素为 0，求积操作的单位元或中性元素为 1。因此，你的求积方法可以如下编写：

```
public static Double product(List<Double> ds) {
```

```
return ds.isEmpty()
    ? 1.0
    : ds.head() * product(ds.tail());
}
```

请注意，求积与求和操作有一个重要的不同。它具有一个满足以下条件的元素，称之为吸收元（absorbing element）：

$a \times \text{吸收元} = \text{吸收元} \times a = \text{吸收元}$

乘法的吸收元为 0。以此类推，任何操作的吸收元（如果存在的话）也被称为零元（zero element）。零元的存在允许你跳出计算，亦称为短路（short circuiting）：

```
public static Double product(List<Double> ds) {
    return ds.isEmpty()
        ? 1.0
        : ds.head() == 0.0
            ? 0.0
            : ds.head() * product(ds.tail());
}
```

不过先抛开这个优化版不谈，看看 sum 和 product 的定义。你能发现一个可以抽象的模式吗？让我们一起来看看（改变参数名之后）：

```
public static Integer sum(List<Integer> list) {
    return list.isEmpty()
        ? 0
        : list.head() + sum(list.tail());
}

public static Double product(List<Double> list) {
    return list.isEmpty()
        ? 1
        : list.head() * product(list.tail());
}
```

现在让我们删除不同并替换为通用标记：

```
public static Type operation(List<Type> list) {
    return list.isEmpty()
        ? identity
        : list.head() operator operation(list.tail());
}

public static Type operation(List<Type> list) {
    return list.isEmpty()
        ? identity
        : list.head() operator operation(list.tail());
}
```

这两个操作几乎完全相同。如果能找到一种抽象公共部分的方法，你就只需提供变量信息 (Type、operation、identity 和 operator) 来实现这两个操作，而不必重复定义。这个常见的操作就是我们说的 *fold*，你在第 3 章中学过了。在那一章中，你学到了两种折叠——右折叠和左折叠——以及这两个操作之间的关系。

清单 5.2 展示了将 *sum* 和 *product* 操作的公共部分抽象到一个名为 *foldRight* 的方法，它接收待折叠的列表、单位元和一个高阶函数（表示用于折叠列表的操作）为参数。单位元显然是给定操作的标识，而函数则是柯里化的形式。（如果你不记得这是什么意思，请参阅第 2 章。）这个函数表示代码的运算符部分。

清单 5.2 实现 *foldRight* 并将其用于 *sum* 和 *product*

```
public static <A, B> B foldRight(List<A> list,      <— A 和 B 表示类型
                                     B n,          <— n 是单位元
                                     Function<A, Function<B, B>> f) {
    return list.isEmpty()
        ? n
        : f.apply(list.head()).apply(foldRight(list.tail(), n, f));
}

public static Integer sum(List<Integer> list) {
    return foldRight(list, 0, x -> y -> x + y);
}

public static Double product(List<Double> list) {
    return foldRight(list, 1.0, x -> y -> x * y);
}
```

f 是表示运算符的函数

sum 和 product 是操作的名称

请注意，Type 变量部分在此被替换为两种类型，A 和 B。这是因为折叠的结果并不总是与列表的元素类型相同。在这里，它抽象得比求和与求积操作所需的更多，不过很快就会派上用场。

operation 变量部分当然是两个方法的名称。

折叠操作并不限于数学计算。你可以使用折叠将字符列表转换为字符串。在这种情况下，A 和 B 是两种不同类型：Char 和 String。但是你也可以使用折叠将字符串列表转换为单个字符串。现在你能想到如何实现 concat 吗？

顺便提一句，*foldRight* 非常类似于单链表。如果把列表 1、2、3 当作

Cons(1, Cons(2, Cons(3, Nil))

你可以立即看到它与右折叠非常相似：

```
f(1, f(2, f(3, identity)))
```

不过可能你已经意识到 Nil 是向列表中添加元素的单位元。这是有道理的：如果想将字符列表转换为字符串，你需要始于一个空列表。（顺便提一句，Nil 也是列表连接的单位元，尽管你可以不用它，只要待连接列表的列表不为空。在这种情况下，它被称为 *reduce* 而不是 *fold*。但这仅仅是因为结果与元素的类型相同罢了。）

可以通过将 Nil 和 cons 传递给 foldRight 作为单位元和用于折叠的函数来实现：

```
List.foldRight(list(1, 2, 3), list(), x -> y -> y.cons(x))
```

这样便简单地生成了一个具有相同元素及相同顺序的新列表，你可以通过运行以下代码看到：

```
System.out.println(List.foldRight(list(1, 2, 3), list(),  
                                  x -> y -> y.cons(x)));
```

这段代码生成以下输出：

```
[1, 2, 3, NIL]
```

以下信息说明了每一步都发生了什么：

```
foldRight(list(1, 2, 3), list(), x -> y -> y.cons(x));  
foldRight(list(1, 2), list(3), x -> y -> y.cons(x));  
foldRight(list(1), list(2, 3), x -> y -> y.cons(x));  
foldRight(list(), list(1, 2, 3), x -> y -> y.cons(x));
```

练习 5.9

编写一个方法来计算列表的长度。该方法将调用 foldRight 方法。

答案 5.9

Nil 的实现显而易见，返回 0 即可。Cons 的实现可以编写为

```
public int length() {  
    return foldRight(this, 0, x -> y -> y + 1);  
}
```

请注意，该实现除了基于栈的递归之外，性能非常差。即便转换为基于堆的实现，它仍然是 $O(n)$ ，意味着返回长度所需的时间与列表的长度成正比。在以下章节中，你将看到如何在常数时间内获取链表的长度。

练习 5.10

`foldRight` 方法使用了递归，但它不是尾递归，因此会迅速栈溢出。溢出的速度取决于几个因素，其中最重要的是栈的大小。在 Java 中，栈的大小可以通过 `-Xss` 命令行参数进行配置，但主要的缺点是所有线程都会使用相同的大小。使用更大的栈会浪费大多数线程的内存。

创建一个尾递归的 `foldLeft` 方法替换 `foldRight`，可以实现栈安全。这是它的签名：

```
public abstract <B> B foldLeft(B identity, Function<B, Function<A, B>> f);
```

提示

如果你不记得 `foldLeft` 和 `foldRight` 之间的区别，请参阅 3.3.5 节。

答案 5.10

显然 `Nil` 实现要返回单位元。对于 `Cons` 实现而言，首先定义供外部使用的方法 `foldLeft`，调用基于栈的尾递归辅助方法 `foldLeft_`，并传入初始化为 `identity` 的累加器 `acc` 和对 `this` 的引用：

```
public <B> B foldLeft(B identity, Function<B, Function<A, B>> f) {
    return foldLeft_(identity, this, f);
}

private <B> B foldLeft_(B acc, List<A> list,
                        Function<B, Function<A, B>> f) {
    return list.isEmpty()
        ? acc
        : foldLeft_(f.apply(acc).apply(list.head()), list.tail(), f);
}
```

然后进行以下更改，以便可以使用第 4 章中定义的 `TailCall` 接口（静态导入了 `ret` 和 `sus` 方法）：

```
public <B> B foldLeft(B identity, Function<B, Function<A, B>> f) {
    return foldLeft_(identity, this, f).eval();
}

private <B> TailCall<B> foldLeft_(B acc, List<A> list,
                                Function<B, Function<A, B>> f) {
    return list.isEmpty()
        ? ret(acc)
        : sus(() -> foldLeft_(f.apply(acc).apply(list.head()),
```

```

    list.tail(), f));
}

```

练习 5.11

使用你的新 `foldLeft` 方法创建全新栈安全版本的 `sum`、`product` 和 `length`。

答案 5.11

这是 `sumViaFoldLeft` 方法：

```

public static Integer sumViaFoldLeft(List<Integer> list) {
    return list.foldLeft(0, x -> y -> x + y);
}

```

`productViaFoldLeft` 方法如下：

```

public static Double productViaFoldLeft(List<Double> list) {
    return list.foldLeft(1.0, x -> y -> x * y);
}

```

这是 `lengthViaFoldLeft` 方法：

```

public static <A> Integer lengthViaFoldLeft(List<A> list) {
    return list.foldLeft(0, x -> ignore -> x + 1);
}

```

请注意，`length` 方法的第二个参数（表示每次在方法递归调用时列表中的每个元素）会被忽略。这个方法与上一个方法的效率一样低，不应该用于生产代码中。

练习 5.12

使用 `foldLeft` 编写一个用于反转列表的静态函数式方法。

答案 5.12

通过左折叠来反转列表非常简单，从作为累加器的空列表开始，并将首个列表中的每个元素 `cons` 到该累加器中：

```

public static <A> List<A> reverseViaFoldLeft(List<A> list) {
    return list.foldLeft(list(), x -> x::cons);
}

```

此示例使用了方法引用而不是 `lambda`，如第 2 章中所述。如果你喜欢使用 `lambda`，则它等价于以下内容：

```

public static <A> List<A> reverseViaFoldLeft(List<A> list) {

```

```
return list.foldLeft(list(), x -> a -> x.cons(a));
}
```

练习 5.13 (难)

通过 `foldLeft` 编写 `foldRight`。

答案 5.13

这个实现可以用于获取一个栈安全版的 `foldRight`：

```
public static <A, B> B foldRightViaFoldLeft(List<A> list,
                                           B identity, Function<A, Function<B, B>> f) {
    return list.reverse().foldLeft(identity, x -> y -> f.apply(y).apply(x));
}
```

请注意，你也可以通过 `foldRight` 来定义 `foldLeft`，虽然这不太有用：

```
public static <A, B> B foldLeftViaFoldRight(List<A> list,
                                           B identity, Function<B, Function<A, B>> f) {
    return List.foldRight(list.reverse(), identity, x -> y ->
                          f.apply(y).apply(x));
}
```

同样请注意，你使用的 `foldLeft` 方法是 `List` 的实例方法。与此相反，`foldRight` 是一个静态方法。（我们很快就会定义一个 `foldRight` 实例方法。）

5.4.1 基于堆的 `foldRight` 递归版

如我所说，`foldRight` 的递归实现仅用于演示这些概念，因为它是基于栈的，因此不应该用于生产代码。还要注意的，这是一个静态实现。实例实现会更容易使用，允许你通过对象标记使用链式方法调用。

练习 5.14

使用第 4 章中学到的内容编写一个基于堆递归的实例版 `foldRight` 方法。

提示

该方法可以在 `List` 父类中定义。编写一个基于栈的尾递归版的 `foldRight` 方法（使用辅助方法）。然后使用在第 4 章中开发的 `TailCall` 接口将辅助方法更改为基于堆的递归实现。

答案 5.14

首先，让我们编写基于栈的尾递归辅助方法。你需要做的全部就是编写一个辅

助方法，它额外接收一个累加器作为参数。累加器具有与函数返回类型相同的类型，并且其初始值等于 identity 元素（顺便说一下，它使用了两次）。

```
public <B> B foldRight_(B acc, List<A> ts, B identity,
                      Function<A, Function<B, B>> f) {
    return ts.isEmpty()
        ? acc
        : foldRight_(f.apply(ts.head()).apply(acc), ts.tail(), identity, f);
}
```

然后编写调用此辅助方法的主方法：

```
public <B> B foldRight(B identity, Function<A, Function<B, B>> f) {
    return foldRight_(identity, this.reverse(), identity, f);
}
```

现在让这两个方法使用基于堆递归的 TailCall：

```
public <B> B foldRight(B identity, Function<A, Function<B, B>> f) {
    return foldRight_(identity, this.reverse(), identity, f).eval();
}

private <B> TailCall<B> foldRight_(B acc, List<A> ts, B identity,
                                   Function<A, Function<B, B>> f) {
    return ts.isEmpty()
        ? ret(acc)
        : sus(() -> foldRight_(f.apply(ts.head()).apply(acc),
                                ts.tail(), identity, f));
}
```

当然，你也应该编写 Nil 的实现，它很简单。

可以通过重用 foldLeft 让你的 foldRight 实现变得更简短：

```
public <B> B foldRight(B identity, Function<A, Function<B, B>> f) {
    return reverse().foldLeft(identity, x -> y -> f.apply(y).apply(x));
}
```

练习 5.15

通过 foldLeft 或 foldRight 来实现 concat。

答案 5.15

可以用右折叠轻松实现 concat 方法：

```
public static <A> List<A> concat(List<A> list1, List<A> list2) {
    return foldRight(list1, list2, x -> y -> new Cons<>(x, y));
}
```

另一个解决办法是使用左折叠。在这种情况下，实现与 `reverseViaFoldLeft` 相同，同样应用于反转的第一个列表，并使用第二个列表作为累加器：

```
public static <A> List<A> concat(List<A> list1, List<A> list2) {
    return list1.reverse().foldLeft(list2, x -> x::cons);
}
```

这个实现（基于 `foldLeft`）似乎效率较低，因为它必须首先反转第一个列表。事实上并非如此，因为 `foldRight` 的实现是基于左折叠反转的列表。如果还不清楚，请参考 `reverse`（练习 5.6）、`foldLeft`（练习 5.10）和 `foldRight`（清单 5.2）的实现。

练习 5.16

编写一个方法，用于将列表的列表展平为包含每个子列表的所有元素的列表。

提示

这个操作由一系列的连接所组成。换句话说，它类似于对整型列表的所有元素求和，其中整型被替换为列表，而加法被替换为连接。除此之外，它与 `sum` 方法完全相同。

答案 5.16

在这个解决方案中，可以使用方法引用以替换 `lambda` 来表示函数的第二部分：

`x -> x::concat` 等价于 `x -> y -> x.concat(y)`。

```
public static <A> List<A> flatten(List<List<A>> list) {
    return foldRight(list, List.<A>list(), x -> y -> concat(x,y));
}
```

5.4.2 映射和过滤列表

你可以定义许多有用的抽象来处理列表。其中之一是通过对列表内的所有元素应用公共函数来改变它们。

练习 5.17

编写一个函数式方法，接收一个整型列表并将其每个元素乘以 3。

提示

尝试使用到目前为止定义的方法。不要显式地使用递归。目标是抽象一个栈安全的递归，让你不必每次都重新实现即可使用。

答案 5.17

```
public static List<Integer> triple(List<Integer> list) {
    return List.foldRight(list, List.<Integer>list(), h -> t ->
        t.cons(h * 3));
}
```

练习 5.18

编写一个函数，将 List<Double> 中的每个值都转换为 String。

答案 5.18

这个操作可以当成是将预期类型 (List<String>) 的空列表与原始列表连接在一起，并将每个元素转换之后 cons 到累加器里。因此，实现与你在 concat 方法中所做的非常类似：

```
public static List<String> doubleToString(List<Double> list) {
    return List.foldRight(list, List.<String>list(),
        h -> t -> t.cons(Double.toString(h)));
}
```

从空列表开始

Cons 已转换元素

练习 5.19

编写一个通用的函数式方法 map，允许你通过向列表中的每个元素应用指定的函数来修改它。这次使它成为 List 的一个实例方法。在 List 类中添加以下声明：

```
public abstract <B> List<B> map(Function<A, B> f);
```

提示

使用 foldRight 方法的栈安全实例版本。

答案 5.19

map 方法可以在 List 父类中实现：

```
public <B> List<B> map(Function<A, B> f) {
    return foldRight(list(), h -> t -> new Cons<>(f.apply(h), t));
}
```

练习 5.20

编写一个 `filter` 方法，从列表中删除不符合给定断言的元素。再次将其实现为实例方法，签名如下：

```
public List<A> filter(Function<A, Boolean> f)
```

答案 5.20

这是一个在 `List` 父类中使用 `foldRight` 的实现。不要忘记用这个方法的栈安全版本。

```
public List<A> filter(Function<A, Boolean> f) {
    return foldRight(list(), h -> t -> f.apply(h) ? new Cons<>(h,t) : t);
}
```

练习 5.21

编写一个 `flatMap` 方法，该方法对 `List<A>` 的每个元素应用一个从 `A` 到 `List` 的函数，并返回一个 `List`。它的签名为

```
public <B> List<B> flatMap(Function<A, List<B>> f);
```

例如，`List.list(1,2,3).flatMap(i -> List.list(i, -i))` 应该返回 `list(1,-1,2,-2,3,-3)`。

答案 5.21

可以再次在 `List` 父类中使用 `foldRight` 将其实现：

```
public <B> List<B> flatMap(Function<A, List<B>> f) {
    return foldRight(list(), h -> t -> concat(f.apply(h), t));
}
```

练习 5.22

基于 `flatMap` 创建一个新版本的 `filter`。

答案 5.22

这是一个静态实现：

```
public static <A> List<A> filterViaFlatMap(List<A> list,
                                           Function<A, Boolean> p) {
    return list.flatMap(a -> p.apply(a) ? List.list(a) : List.list());
}
```

请注意，`map`、`flatten` 和 `flatMap` 之间存在很强的关系。如果映射一个从列表到列表的函数，则会得到列表的列表。你可以随后应用 `flatten` 以获取包含所有子列表内的元素的单一列表。直接应用 `flatMap` 也可以得到完全相同的结果。

这种关系的一个结果就是你可以用 `flatMap` 重新定义 `flatten`：

```
public static <A> List<A> flatten(List<List<A>> list) {  
    return list.flatMap(x -> x);  
}
```

这并不在意料之外，因为调用 `concat` 已经被抽象为 `flatMap` 了。

5.5 总结

- 数据结构是编程中最重要的概念之一。
- 单链表是函数式编程中最常用的数据结构。
- 使用不可变和持久化列表可以获得线程安全。
- 使用数据共享允许大多数操作（虽然不是所有）拥有极高性能。
- 可以创建其他的数据结构，以获得针对特定用例的良好性能。
- 可以通过递归应用函数来折叠列表。
- 可以使用基于堆的递归来折叠列表，而不会有栈溢出的风险。
- 一旦定义了 `foldRight` 和 `foldLeft`，就不再需要使用递归来处理列表。
`foldRight` 和 `foldLeft` 抽象了递归。

6 处理可选数据

本章要点

- null 引用，或者说“十亿美元的错误”
- null 引用的替代方法
- 为可选数据（optional data）开发 Option 数据类型
- 将函数应用于可选值
- 复合可选值
- Option 的用例

在计算机程序中表示可选数据一直是一个问题。在日常生活中，可选数据的概念非常简单。当某个东西被包含在容器中时，很容易就可以表示这个东西缺失——无论是什么，它都可以由空容器来表示。一个空苹果篮子可以表示没有苹果，一个空油箱可以当作汽车中没有汽油。

表示计算机程序中的缺失数据却很困难。大部分数据通过指向它的引用来表示，因此最明显的表示缺失数据的方法是使用一个指向虚无的指针。这就是空指针（null pointer）的由来。

在 Java 中，变量是指向值的指针。变量可以被创建为 null（创建静态和实例

变量的默认值为 `null`)，然后可以把它改为指向值。如果数据被删除，它们甚至还可以再改回指向 `null`。

为了处理可选数据，在 Java 8 中引入了 `Optional` 类型。尽管如此，你将在本章中开发自己的类型，并称其为 `Option`。这样做的目的是了解这种结构如何工作。完成本章后，你就可以随意使用标准 Java 8 版本库的 `Optional` 了，不过在接下来的章节中，你将会看到它比你本章中所创建的类型弱得多。

6.1 空指针的问题

`NullPointerException` 是命令式程序中最常见的 bug 之一。当一个标识被取消引用并且没有指向任何内容时，就会引发此错误。换句话说，预期一些数据存在，但它却缺失了。这样的标识称为指向 `null`。

`null` 引用是由 Tony Hoare 在 1965 年发明的，那时候他在设计面向对象的语言 ALGOL。以下是他 44 年之后的话：¹

我称之为十亿美元的错误……我当时的目标是确保所有引用的使用都应该绝对安全，由编译器自动执行检查。但是我无法抵制引入一个空引用的诱惑，仅仅是因为它很容易实现。这导致了无数的错误、漏洞和系统崩溃，它很可能在过去的四十年中造成了十亿美元的痛苦和损失。

虽然如今避免空引用是众所周知的，但是事实远非如此。Java 的标准库中包含了接收可选参数的方法和构造函数，如果未使用，必须将其设置为 `null`。举一个例子，`java.net.Socket` 类。它定义了以下构造函数：

```
public Socket(String address,
               int port,
               InetAddress localAddr,
               int localPort) throws IOException
```

根据文档，

如果指定的本地地址为 `null`，则相当于将地址指定为任意的本地地址。

`null` 引用在此是有效的参数。有时这被称为 *business null*（业务 `null`）。请注意，这种处理缺失数据的方式并不只特定于对象。端口也可以不存在，但却不能为 `null`，

¹ Tony Hoare, “空引用：十亿美元的错误” (QCon, 2009年8月25日), <http://mng.bz/12MC>。

因为它是原始类型：

本地端口号为 0 会让系统在绑定操作中使用一个空闲端口。

这种值有时被称为 *sentinel value* (标记值)。它并不使用值本身 (并不表示端口为 0)，而是用于规定端口值的缺失。

Java 库中还有许多处理缺失数据的例子。这是非常危险的，因为事实上本地地址为 null 可能是由于先前的错误而无意招致的。但它并不会造成异常，程序还将按照意料之外的方式继续工作。

还有其他情况的业务 null。如果你尝试使用不存在的键从 HashMap 中取值，将得到一个 null。这是一个错误吗？你不知道。有可能键是有效的，但并没有在 map 中注册；也有可能假定键是有效的，应该在 map 中，但是先前计算出的键有误。例如，无论是有意为之还是百密一疏，键可以为 null，并且不会引发异常。它甚至还可以返回一个非 null 值，因为在 HashMap 中允许键为 null。这样的情况真是一团糟。

当然，你知道应该怎么做。你知道不检查引用是否为 null 就不应该使用它。(对方法接收的每个对象参数你都会这样做，不是吗？) 你知道不先测试 map 中是否包含相应的键就不应该从 map 中取值。你知道不应该试图从列表中取得一个元素，而不首先确认列表不为空，还有如果通过索引访问元素，需要确保列表中有足够的元素。你一直这样做，所以你永远也不会得到一个 NullPointerException 或 IndexOutOfBoundsException。

如果你是这种完美的程序员，你可以使用空引用。但是对我们其余的人来说，有必要引入更简单和更安全的方法来处理值的缺失，无论是故意的还是由于错误导致的。在本章中，你将学习如何处理并非由错误导致的缺失值。这种数据被称为可选数据。

一直存在着处理可选数据的技巧。最著名和最常用的当属列表。当一个方法预期返回一个值或没有返回值时，有些程序员使用列表作为返回值。列表可以包含零个或一个元素。虽然可以完美地工作，但是它也有几个重要的缺点：

- 没有办法确保列表最多包含一个元素。如果接收到含有多个元素的列表，你应该怎么办？
- 如何区分应该包含最多一个元素的列表和常规列表？
- List 类定义了许多方法和函数来处理列表可能包含多个元素的事实。这些

方法对我们来说毫无用处。

- 函数式列表是递归结构，而你并不需要它。更简单的实现就足够了。

6.2 空引用的替代方案

看起来我们的目标像是要避免 `NullPointerException`，但这并不准确。`NullPointerException` 应该总是表示一个 bug。因此，你应该应用“快速失败”原则：如果有错误，程序应该尽快失败。完全删除业务 `null` 不会让你摆脱 `NullPointerException`，它只能确保 `null` 引用只会由程序中的 bug 而非可选数据所引起。

以下代码是一个返回可选数据的示例方法：

```
static Function<List<Integer>, Double> mean = xs -> {  
    if (xs.isEmpty()) {  
        ???;  
    } else {  
        return xs.foldLeft(0.0, x -> y -> x + y) / xs.length();  
    }  
};
```

`mean` 函数是一个偏函数的例子，正如第 2 章中所见：它定义了除空列表之外的所有列表。你应该如何处理空列表的情况？

返回一个标记值是一种可选方案。你应该选择什么值？因为类型为 `Double`，所以你可以使用在 `Double` 类中定义的值：

```
static Function<List<Integer>, Double> mean = xs -> {  
    if (xs.isEmpty()) {  
        return Double.NaN;  
    } else {  
        return xs.foldLeft(0.0, x -> y -> x + y) / xs.length();  
    }  
};
```

因为 `Double.NaN`（不是数字）实际上是一个 *double* 值（注意这是小写字母 d），所以这样做能够工作。`Double.NaN` 是一个原始类型！

到目前为止还算不错，不过你还有三个问题：

- 如果想对一个返回 `Integer` 的函数故伎重演应该怎么做？整型类里没有等价于 `NaN` 的值。

- 如何告知你的函数的用户它可以返回一个标记值？
- 如何处理泛型函数（parametric function），例如

```
static <A, B> Function<List<A>, B> f = xs -> {
    if (xs.isEmpty()) {
        ???;
    } else {
        return ...;
    };
};
```

另一个解决方案是抛出异常：

```
static Function<List<Integer>, Double> mean = xs -> {
    if (xs.isEmpty()) {
        throw new MeanOfEmptyListException();
    } else {
        return xs.foldLeft(0.0, x -> y -> x + y) / xs.length();
    }
};
```

但这个方案是丑陋的，并且制造出的麻烦比解决的麻烦更多：

- 异常通常用于错误的结果，但是这里没有错误。没有结果只是因为没有任何输入数据！或许你应该认为传入空列表来调用函数是一个 bug？
- 你应该抛出什么异常？一个自定义的（如示例所示）？还是一个标准的？
- 应该使用可控异常（checked exception）还是不可控异常（unchecked exception）？此外，你的函数不再是一个纯函数。它不再是引用透明的，这将导致我在第2章中谈过的许多问题。此外，你的函数不能再复合了。

你也可以返回 null，让调用者来处理它：

```
static Function<List<Integer>, Double> mean = xs -> {
    if (xs.isEmpty()) {
        return null;
    } else {
        return xs.foldLeft(0.0, x -> y -> x + y) / xs.length();
    }
};
```

返回 null 是最糟糕的方案：

- 它强制（理想地）调用者检查结果是否为 null 并采取相应行动。
- 如果使用了装箱（boxing）就会崩溃。

- 与异常的方案一样，函数不能再复合了。
- 它允许潜在的问题扩散至远离源头。如果调用者忘记检查结果是否为 null，则代码中的任何地方都有可能抛出 NullPointerException。

一个更好的解决方案是请用户提供一个特殊的值，如果没有可用数据则将其返回。例如，以下函数计算列表的最大值：

```
static <A, B> Function<B, Function<List<A>, B>> max = x0 -> xs -> {
    return xs.isEmpty()
        ? x0
        : ...;
```

下面这段代码介绍如何定义一个 max 函数：

```
static <A extends Comparable<A>> Function<A, Function<List<A>, A>> max() {
    return x0 -> xs -> xs.isEmpty()
        ? x0
        : xs.tail().foldLeft(xs.head(), x -> y -> x.compareTo(y) < 0 ? x : y);
}
```

请记住，你必须使用一个返回函数的方法，因为属性无法参数化。

如果你觉得这样太复杂了，这里有一个函数式方法的版本：

```
static <A extends Comparable<A>> A max(A x0, List<A> xs) {
    return xs.isEmpty()
        ? x0
        : xs.tail().foldLeft(xs.head(), x -> y -> x.compareTo(y) < 0 ? x : y);
}
```

这可以工作，但是过于复杂了。最简单的办法是返回一个列表：

```
public static <A extends Comparable<A>> Function<List<A>, List<A>> max() {
    return xs -> xs.isEmpty()
        ? List.list()
        : List.list(xs.foldLeft(xs.head(), x -> y -> x.compareTo(y) < 0
            ? x : y));
}
```

虽然这个解决方案能够完美工作，但是它有点丑陋，参数类型和函数的返回类型相同，虽然它们并不代表相同的东西。为了解决这个问题，你可以简单地创建一个类似于 List 的新类型，但使用不同的名字来表示它的含义。当用它工作时，你可以选择一个更合适的实现，来确保这个“列表”至多有一个元素。

6.3 Option数据类型

你在本章中创建的 Option 数据类型将与 List 数据类型非常类似。为可选数据使用 Option 类型，甚至允许你在数据缺失时也能够复合函数（见图 6.1）。它将被实现为一个 Option 抽象类，包含两个私有子类，分别表示数据存在以及缺失。表示数据缺失的子类称为 None，而表示数据存在的子类称为 Some。一个 Some 子类将包含相应的数据值。

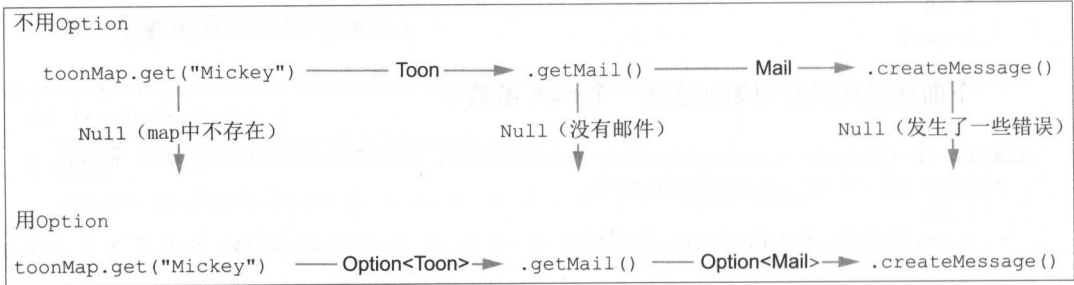


图 6.1 没有 Option 类型，复合函数不会生成一个函数，因为表示结果的程序可能会抛出 NullPointerException。

清单 6.1 展示了这三个类的代码。

清单 6.1 Option 数据类型

```
package optionaldata;

public abstract class Option<A> {

    @SuppressWarnings("rawtypes")
    private static Option none = new None();
    public abstract A getOrThrow();

    private static class None<A> extends Option<A> {

        private None() {}

        @Override
        public A getOrThrow() {
            throw new IllegalStateException("get called on None");
        }

        @Override
        public String toString() {
            return "None";
        }
    }
}
```

None 的单例实例将用于所有类型

getOrThrow() 允许你从 Option 中检索值

构造函数是私有的

None 子类表示值缺失

在 None 类中，getOrThrow() 抛出异常

toString() 返回人类可读的字符串以表示一个 Option

```

}

private static class Some<A> extends Option<A> {

    private final A value;

    private Some(A a) {
        value = a;
    }

    @Override
    public A getOrThrow() {
        return this.value;
    }

    @Override
    public String toString() {
        return String.format("Some(%s)", this.value);
    }
}

public static <A> Option<A> some(A a) {
    return new Some<>(a);
}

@SuppressWarnings("unchecked")
public static <A> Option<A> none() {
    return none;
}
}

```

构造函数是私有的

toString() 返回人类可读的字符串以表示一个 Option

some 工厂方法允许你用一个值创建一个 Option

none 工厂方法返回 none 单例

在清单 6.1 中，你可以看到 Option 与 List 有多接近。它们都是具有两个私有实现的抽象类。None 子类对应于 Nil，Some 子类对应于 Cons。getOrThrow 方法类似于 List 中的 head 方法。

你可以用 Option 来定义 max 函数，如下所示：

```

static <A extends Comparable<A>> Function<List<A>, Option<A>> max() {
    return xs -> xs.isEmpty()
        ? Option.none()
        : Option.some(xs.foldLeft(xs.head(),
            x -> y -> x.compareTo(y) > 0 ? x : y));
}

```

现在你的函数是一个全函数，这意味着它对包括空列表的所有列表都有一个值。注意，这段代码与返回列表的版本是多么相似。虽然 Option 的实现与 List 的实现不同，但它们的用法几乎完全相同。正如你将要看到的，相似性远不止于此。

但实际上, `Option` 类不是非常有用。使用 `Option` 的唯一方式就是检查实际的类,看看它是一个 `Some` 还是一个 `None`,并调用 `getOrThrow` 方法来获得前者的值。如果没有数据,这个方法将抛出异常,这不太函数式。为了使它成为一个强大的工具,你需要添加一些方法,就像先前在 `List` 中所做的一样。

6.3.1 从 `Option` 中取值

你为 `List` 创建的许多方法对 `Option` 也有用。事实上,只有像折叠那样与多个值相关的方法,可能在这里才没有用处。但在创建这些方法之前,让我们从一些特定于 `Option` 的用法开始。

为了避免检查 `Option` 的子类,你需要定义与 `getOrThrow` 不同的方法,它可能在两个子类中都有用,所以你可以在 `Option` 父类中调用它们。你需要的第一个方法是获取 `Option` 中的值。使用默认值是数据缺失时的一个常见用例。

练习 6.1

实现 `getOrElse` 方法,如果包含的值存在,则将其返回,否则返回一个提供的默认值。这是方法签名:

```
A getOrElse(A defaultValue)
```

答案 6.1

这个方法将实现为一个实例方法,在 `Option` 抽象类中如下声明:

```
public abstract A getOrElse(A defaultValue);
```

很明显, `Some` 的实现应该简单地返回它所包含的值:

```
public A getOrElse(A defaultValue) {  
    return this.value;  
}
```

而 `None` 的实现即返回默认值:

```
public A getOrElse(A defaultValue) {  
    return defaultValue;  
}
```

到目前为止还算不错。你现在可以定义返回 `Option` 的方法并直接使用返回值,如下所示:

```
int max1 = max().apply(List.<Integer>list(3, 5, 7, 2, 1)).getOrElse(0);
int max2 = max().apply(List.list()).getOrElse(0);
```

这里的 max1 将等于 7（列表中的最大值），max2 将被设置为 0（默认值）。

但你可能会有问题。看看以下例子：

```
int max1 = max().apply(List.list(3, 5, 7, 2, 1)).getOrElse(getDefault());
System.out.println(max1);
int max2 = max().apply(List.<Integer>list()).getOrElse(getDefault());
System.out.println(max2);

int getDefault() {
    throw new RuntimeException();
}
```

方法完全不是函数式的，这只是为了告诉你发生了什么。这个例子将会打印出什么？如果你认为它会打印 7 然后抛出异常，那就再思考思考。

由于 Java 是一门严格的语言，这个例子将会直接抛出异常，而不会打印任何内容。方法参数会在实际执行方法之前被计算，无论是否需要它们。因此，无论是在 Some 还是 None 上调用，在任何情况下都会计算 getOrElse 方法的参数。Some 不需要方法参数其实无关紧要。当参数为字面量（literal）时，这没有什么区别，但当它调用一个方法时，就会产生巨大的差异。在任何情况下，getDefault 方法都会被调用，因此第一行将抛出异常并且不会显示任何内容。通常这不是你想要的。

练习 6.2

通过对 getOrElse 方法的参数使用惰性求值来修复先前的问题。

提示

使用你在第 3 章（练习 3.2）中定义的 Supplier 类。

答案 6.2

方法的签名将更改为

```
public abstract A getOrElse(Supplier<A> defaultValue);
```

除了方法签名以外，Some 的实现并不会改变（由于不使用参数）：

```
@Override
public A getOrElse(Supplier<A> defaultValue) {
```

```

    return this.value;
}

```

最重要的变化在 `None` 类中：

```

@Override
public A getOrElse(Supplier<A> defaultValue) {
    return defaultValue.get();
}

```

如果没有值，则通过调用 `Supplier.get()` 方法来对参数求值。`max` 的例子现在可以如下重写：

```

int max1 = max().apply(List.list(3, 5, 7, 2, 1))
    .getOrElse(() -> getDefault());
System.out.println(max1);
int max2 = max().apply(List.<Integer>list()).getOrElse(() -> getDefault());
System.out.println(max2);
int getDefault() {
    throw new RuntimeException();
}

```

该程序在抛出异常之前将 7 打印到控制台。

有了 `getOrElse` 方法，现在你不再需要 `getOrThrow` 方法了。但是在为 `Option` 类开发其他方法时它还可能会派上用场，因此我们先留着它并将其设为 `protected`。

6.3.2 将函数应用于可选值

`List` 中一个非常重要的方法是 `map` 方法，它允许你将从 `A` 到 `B` 的函数应用到 `A` 列表内的每个元素，从而生成一个 `B` 列表。考虑到 `Option` 类似于最多包含一个元素的列表，你可以应用相同的原理。

练习 6.3

创建一个 `map` 方法，通过应用从 `A` 到 `B` 的函数将 `Option<A>` 更改为 `Option`。

提示

在 `Option` 类中定义一个抽象方法，每个子类都有一个实现。`Option` 中的方法签名是

```
public abstract <B> Option<B> map(Function<A, B> f)
```

答案 6.3

None 的实现很简单。你只需返回一个 None 实例即可。正如我先前所说，Option 类包含了一个可用的 None 单例：

```
public <B> Option<B> map(Function<A, B> f) {  
    return none();  
}
```

请注意，虽然 this 和 none 都引用了同一个对象，但是由于用参数化 A 的原因而不能返回 this。none 指向同一对象，但具有原始类型（无参）。这就是为什么使用 @SuppressWarnings("rawtypes") 注解可避免将编译器的警告泄露给调用者。同样你并不直接访问 none 实例，而是调用 none() 的工厂方法，以避免在 none() 方法中已经通过 @SuppressWarnings("unchecked") 注释避免了“未检查赋值警告（Unchecked assignment warning）”。

Some 的实现不会更复杂。你需要做的全部就是取值、应用该函数，并将结果包装在一个新的 Some 里：

```
public <B> Option<B> map(Function<A, B> f) {  
    return new Some<>(f.apply(this.value));  
}
```

6.3.3 复合 Option 处理

你很快就会意识到，函数式编程中最常见的并不是从 A 到 B 的函数。首先，你可能不那么容易熟悉返回可选值的函数。毕竟，似乎封装 Some 实例中的值并在以后获取它们需要额外的工作。但是随着进一步的练习，你会发现这些操作很少发生。当链接函数构建一个复杂的计算时，你经常会从先前计算返回的一个值开始，将结果传递给一个新的函数，而不会看到中间结果。换句话说，你会更经常使用从 A 到 Option 的函数而不是从 A 到 B 的函数。

想想 List 类。有没有引起共鸣？是的，它引出了 flatMap 方法。

练习 6.4

创建一个 flatMap 实例方法，接收一个从 A 到 Option 的函数为参数，并

返回一个 `Option`。

提示

可以在两个子类中定义不同的实现，但是你应该尝试设计一个独一无二的适用于这两个子类的实现，并将其置于 `Option` 类中。它的签名是

```
<B> Option<B> flatMap(Function<A, Option<B>>> f)
```

尝试使用一些已有的方法（`map` 和 `getOrElse`）。

答案 6.4

一个简单的方案就是在 `Option` 类中定义一个抽象方法，在 `None` 类中返回 `none()`，并在 `Some` 类中返回 `f.apply(this.value)`。这可能是最高效的实现。但是更优雅的方案是映射 `f` 函数，给定一个 `Option<Option>`，然后使用 `getOrElse` 方法取值（`Option`）并传入 `None` 作为默认值：

```
public <B> Option<B> flatMap(Function<A, Option<B>>> f) {
    return map(f).getOrElse(Option::none);
}
```

练习 6.5

正如需要一个办法来映射返回一个 `Option`（引出 `flatMap`）的函数，你将需要一个 `getOrElse` 的版本作为 `Option` 的默认值。使用以下签名创建 `orElse` 方法：

```
Option<A> orElse(Supplier<Option<A>> defaultValue)
```

提示

正如你可能从名称中猜到的，没有必要“`get`”值来实现这个方法。这正是 `Option` 的主要用法：复合 `Option` 而不是包装和取值。结果是两个子类都将适用相同的实现。

答案 6.5

解决方案包含映射函数 `x -> this` 以产生一个 `Option<Option<A>`，然后使用这个结果的 `getOrElse` 并传入默认值：

```
public Option<A> orElse(Supplier<Option<A>> defaultValue) {
    return map(x -> this).getOrElse(defaultValue);
}
```

练习 6.6

在第 5 章中，你创建了一个 `filter` 方法，用于从列表中删除所有不满足条件的元素，条件通过断言（换句话说，它是一个返回 `Boolean` 的函数）的形式来表达。为 `Option` 创建相同的方法。这是它的签名：

```
Option<A> filter(Function<A, Boolean> f)
```

提示

由于 `Option` 像是一个最多只有一个元素的 `List`，这个实现似乎微不足道。在 `None` 子类中，你只需返回 `none()` 即可。在 `Some` 类中，如果条件成立，则返回原来的 `Option`，否则返回 `none()`。不过请尝试设计一个适用于 `Option` 父类的更灵巧的实现。

答案 6.6

解决办法就是对用于 `Some` 的函数应用 `flatMap`：

```
public Option<A> filter(Function<A, Boolean> f) {  
    return flatMap(x -> f.apply(x)  
        ? this  
        : none());  
}
```

6.3.4 Option 的用例

如果了解过 Java 8 的 `Optional` 类，你可能已经注意到 `Optional` 包含一个 `isPresent()` 方法，允许你检查 `Optional` 中是否包含值。（`Optional` 有一个不同的实现，并不是基于两个不同的子类。）你可以很容易地实现这样的方法，不过你会称之为 `isSome()`，因为它会检查对象是一个 `Some` 还是一个 `None`。你也可以称之为 `isNone()`，这似乎更加合乎逻辑，因为它相当于 `List.isEmpty()` 方法。

虽然 `isSome()` 方法有时很有用，但它并不是使用 `Option` 类的最佳做法。如果你在调用 `getOrThrow()` 取值之前通过 `isSome()` 方法来检查一个 `Option`，那与在解引用（`dereferencing`）之前检查引用是否为 `null` 并没有太大的不同。唯一的区别是在忘记首先检查的情况下：你面临的风险是看到一个 `IllegalStateException`，而不是一个 `NullPointerException`。

最佳做法是使用复合 `Option`。为了做到这一点，你必须为所有用例创建所有

必需的方法。这些用例对应于你在检查值是否为 `null` 之后要用它做的事。你可以执行以下操作之一：

- 用该值作为另一个函数的输入。
- 对该值应用作用。
- 当该值不为 `null` 时使用该值，或者使用默认值来应用一个函数或一个作用。

第一个和第三个用例已经可以通过你创建过的方法实现了。应用作用能够以不同的方式完成，你将在第 13 章中学习到相关知识。

举一个例子，看看如何用 `Option` 类来改变你使用 `map` 的方式。清单 6.2 展示了一个函数式 `Map` 的实现。这不是一个函数式实现，而只是一个传统的 `ConcurrentHashMap` 的包装，给它一个函数式接口。

清单 6.2 在函数式 `Map` 中使用 `Option`

```
import com.fpinjava.optionalddata.exercise06_05.Option;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;

public class Map<T, U> {

    private final ConcurrentMap<T, U> map = new ConcurrentHashMap<>();

    public static <T, U> Map<T, U> empty() {
        return new Map<>();
    }

    public static <T, U> Map<T, U> add(Map<T, U> m, T t, U u) {
        m.map.put(t, u);
        return m;
    }

    public Option<U> get(final T t) {
        return this.map.containsKey(t)
            ? Option.some(this.map.get(t))
            : Option.none();
    }

    public Map<T, U> put(T t, U u) {
        return add(this, t, u);
    }

    public Map<T, U> removeKey(T t) {
        this.map.remove(t);
    }
```

← 此版本的 `map` 封装了“使用前检查”的模式，以避免返回 `null` 引用

```

    return this;
}
}

```

如你所见，Option 允许在调用 get 之前，把用 containsKey 查询 map 的模式封装进 map 的实现。清单 6.3 展示了应该如何使用它。

清单 6.3 使用 Option

```

import com.fpinjava.optionalddata.exercise06_06.Option;
import com.fpinjava.optionalddata.listing06_02.Map;

public class UseMap {

    public static void main(String[] args) {

        Map<String, Toon> toons = new Map<String, Toon>()
            .put("Mickey", new Toon("Mickey", "Mouse", "mickey@disney.com"))
            .put("Minnie", new Toon("Minnie", "Mouse"))
            .put("Donald", new Toon("Donald", "Duck", "donald@disney.com"));

        Option<String> mickey = toons.get("Mickey").flatMap(Toon::getEmail);
        Option<String> minnie = toons.get("Minnie").flatMap(Toon::getEmail);
        Option<String> goofy = toons.get("Goofy").flatMap(Toon::getEmail);

        System.out.println(mickey.getOrElse(() -> "No data"));
        System.out.println(minnie.getOrElse(() -> "No data"));
        System.out.println(goofy.getOrElse(() -> "No data"));
    }

    static class Toon {

        private final String firstName;
        private final String lastName;
        private final Option<String> email;

        Toon(String firstName, String lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
            this.email = Option.none();
        }

        Toon(String firstName, String lastName, String email) {
            this.firstName = firstName;
            this.lastName = lastName;
            this.email = Option.some(email);
        }
    }
}

```

通过 flatMap
复合 Option


```

public Option<String> getEmail() {
    return email;
}
}
}

```

在这段（非常简化的）程序中，你可以看到返回 `Option` 的各种函数是如何复合的。你不必检查任何东西，而且没有抛出 `NullPointerException` 的风险，尽管在 `Toon` 没有电子邮件时可能会要求你提供，甚至 `Toon` 不存在于这个 `map` 中。

但是有一个小问题。这段程序打印了

```

mickey@disney.com
No data
No data

```

第一行是 `Mickey` 的电子邮件。第二行因为 `Minnie` 没有电子邮件而显示 “No data”。第三行因为 `Goofy` 不在 `map` 中而显示 “No data”。显然，你需要一种方式来区分这两种情况。`Option` 类不允许你区分它们。你将在下一章中看到如何解决这个问题。

练习 6.7

用 `flatMap` 实现 `variance` 函数。一组值的方差（`variance`）表示这些值如何分布在平均值周围。如果所有值都非常接近平均值，则方差较低。当所有值都等于平均值时，方差为 0。一组方差是 `Math.pow(x - m, 2)` 的平均值，`x` 为该组中每个元素的值，`m` 为该组值的平均值。这是函数的签名：

```
Function<List<Double>, Option<Double>> variance = ...
```

提示

要实现这个函数，必须首先实现一个函数来计算 `List<Double>` 的和。然后你应该像在本章前面介绍的那样创建一个 `mean` 函数，但是使用双精度。如果在定义这些函数时遇到问题，请参阅第 4 章和第 5 章或使用以下函数：

```

static Function<List<Double>, Double> sum =
    ds -> ds.foldLeft(0.0, a -> b -> a + b);

static Function<List<Double>, Option<Double>> mean =
    ds -> ds.isEmpty()
        ? Option.none()
        : Option.some(sum.apply(ds) / ds.length());

```

答案 6.7

一旦定义了 `sum` 和 `mean` 函数，`variance` 函数就相当简单了：

```
static Function<List<Double>, Option<Double>> variance =
    ds -> mean.apply(ds)
        .flatMap(m -> mean.apply(ds.map(x -> Math.pow(x - m, 2))));
```

请注意，并不强制使用函数。如果需要将它们作为参数传递给高阶函数，则必须使用函数，但如果只需要应用它们，函数式方法可能会更加易于使用。

如果你希望尽可能地使用方法，可以实现如下方案：

```
public static Double sum(List<Double> ds) {
    return sum_(0.0, ds).eval();
}

public static TailCall<Double> sum_(Double acc, List<Double> ds) {
    return ds.isEmpty()
        ? ret(acc)
        : sus(() -> sum_(acc + ds.head(), ds.tail()));
}

public static Option<Double> mean(List<Double> ds) {
    return ds.isEmpty()
        ? Option.none()
        : Option.some(sum(ds) / ds.length());
}

public static Option<Double> variance(List<Double> ds) {
    return mean(ds).flatMap(m -> mean(ds.map(x -> Math.pow(x - m, 2))));
}
```

如你所见，有两个原因让函数式方法更易使用。首先，你不需要在函数名和参数之间编写 `.apply`。其次，由于不需要写下 `Function`，因此类型更加简短。正因如此，你应该尽量使用函数式方法而非函数。

但请记住，它们之间的切换很容易。给定以下方法，

```
B aToBmethod(A a) {
    return ...
}
```

你可以通过编写以下代码创建一个等价函数：

```
Function<A, B> aToBfunction = a -> aToBmethod(a);
```

你也可以使用方法引用：

```
Function<A, B> aToBfunction = this::aToBmethod;
```

反过来，你可以由上述函数创建一个方法：

```
B aToBmethod2(A a) {
    return aToBfunction.apply(a)
}
```

就像 `variance` 的实现演示的那样，你可以使用 `flatMap` 来构造一个多阶段的计算，其中的任何一个阶段都可能会失败，而计算将在遇到第一个失败时立即中止，因为 `None.flatMap(f)` 将立即返回 `None` 而不会应用 `f`。

6.3.5 复合 Option 的其他方法

决定使用 `Option` 似乎带来了巨大的影响。尤其是一些开发人员可能相信他们的遗留代码将被废弃。如果需要从一个 `Option<A>` 到 `Option` 的函数，而你只有一个 API 用方法将 `A` 转换为 `B`，那你现在能够做些什么呢？需要重写所有的库吗？完全不是。你可以很容易改写它们。

练习 6.8

定义一个 `lift` 方法，它接收从 `A` 到 `B` 的函数为参数，并返回一个从 `Option<A>` 到 `Option` 的函数。与往常一样，使用你已经定义过的方法。图 6.2 展示了 `lift` 方法的工作方式。

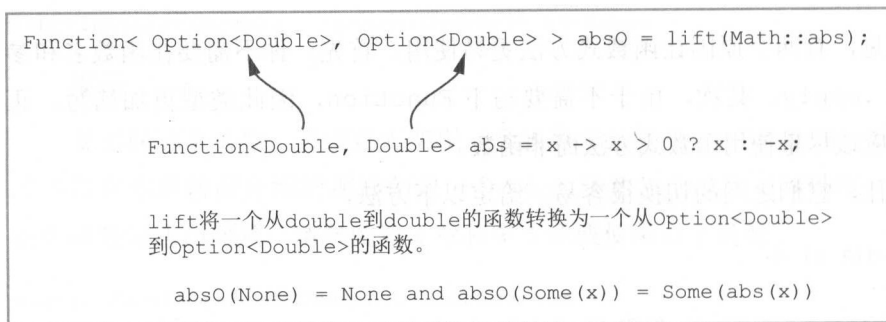


图 6.2 提升 (lift) 一个函数。

提示

用 `map` 方法在 `Option` 类中创建一个静态方法。

答案 6.8

解决方案很简单：

```
static <A, B> Function<Option<A>, Option<B>> lift(Function<A, B> f) {
    return x -> x.map(f);
}
```

当然，现有的大多数库包含的不是函数而是方法。将接收 A 为参数并返回 B 的方法转换为从 Option<A> 到 Option 的函数很容易。例如，可以这样提升 String.toUpperCase 方法：

```
Function<Option<String>, Option<String>> upperOption =
    lift(x -> x.toUpperCase());
```

也可以使用一个方法引用：

```
Function<Option<String>, Option<String>> upperOption =
    lift(String::toUpperCase);
```

练习 6.9

这样的解决方案对抛出异常的方法无效。编写一个 lift 方法，使其适用于抛出异常的方法。

答案 6.9

你需要做的全部就是在 lift 返回函数的实现中包装一个 try...catch 块，如果抛出异常则返回 None：

```
static <A, B> Function<Option<A>, Option<B>> lift(Function<A, B> f) {
    return x -> {
        try {
            return x.map(f);
        } catch (Exception e) {
            return Option.none();
        }
    };
}
```

你可能还需要将一个从 A 到 B 的函数转换为一个从 A 到 Option 的函数。可以应用相同的技术：

```
static <A, B> Function<A, Option<B>> hlift(Function<A, B> f) {
    return x -> {
        try {
```

```

    return Option.some(x).map(f);
  } catch (Exception e) {
    return Option.none();
  }
};
}

```

然而请注意，这并不是非常有效，因为异常丢失了。你将在下一章中学习如何解决这个问题。

如果你打算用接收两个参数的遗留方法那该怎么办呢？假设你要通过一个 `Option<String>` 和一个 `Option<Integer>` 使用 `Integer.parseInt(String s, int radix)`。那该怎么做？

第一步是由这个方法创建一个函数。这很简单：

```

Function<Integer, Function<String, Integer>> parseWithRadix =
    radix -> string -> Integer.parseInt(string, radix);

```

请注意，我在此反转了参数以创建一个柯里化函数。这是有意义的，因为只应用于 `radix` 能够给我们一个有用的函数，即可以通过一个给定的 `radix` 来解析所有的字符串：

```

Function<String, Integer> parseHex = parseWithRadix.apply(16);

```

反之（首先应用于 `String`），意义就不太大了。

练习 6.10

编写一个 `map2` 方法，接收 `Option<A>`、`Option` 和从 `(A, B)` 到 `C` 的柯里化形式的函数为参数，并返回一个 `Option<C>`。

提示

使用 `flatMap`，可能还有 `map` 方法。

答案 6.10

这是使用 `flatMap` 和 `map` 的解决方案。你会经常遇到这种模式，理解它非常重要。我们将在第 8 章再回来讨论。

```

<A, B, C> Option<C> map2(Option<A> a,
    Option<B> b,
    Function<A, Function<B, C>> f) {

```

```
return a.flatMap(ax -> b.map(bx -> f.apply(ax).apply(bx)));
}
```

通过 map2, 你现在可以使用任何双参方法, 就像它是为操作 Option 而创建的。

如果有更多参数的方法呢? 这里有一个 map3 方法的例子:

```
<A, B, C, D> Option<D> map3(Option<A> a,
                             Option<B> b,
                             Option<C> c,
                             Function<A, Function<B, Function<C, D>>> f) {
return a.flatMap(ax -> b.flatMap(bx -> c.map(cx ->
                                             f.apply(ax).apply(bx).apply(cx))));
}
```

你找到规律了吗?

6.3.6 复合 Option 和 List

复合 Option 实例不是你要的全部。你定义的每个新类型都需要在一些时候复合任何其他类型。在第 5 章中, 你定义了 List 类型。为了编写实用的程序, 你需要能够复合 List 和 Option。

最常见的操作是将 List<Option<A>> 转换为 Option<List<A>>。List<Option<A>> 是在用从 B 到 Option<A> 的函数映射 List 时得到的。一般来说, 如果所有的元素都是 Some<A>, 你要的结果就会是一个 Some<List<A>>; 如果至少有一个元素是 None<A>, 则是 None<List<A>>。

练习 6.11

编写一个将 List<Option<T>> 转换为一个 Option<List<T>> 的 sequence 函数。如果原始列表中的所有值都是 Some 实例, 则为 Some<List<T>>, 否则为 None<List<T>>。这是它的签名:

```
Option<List<A>> sequence(List<Option<A>> list)
```

提示

为了找到方向, 你可以检查列表来查看它是否为空, 并且在不为空时递归调用 sequence。记得 foldRight 和 foldLeft 抽象了递归, 接下来, 你可以使用这些方法之一来实现 sequence。

答案 6.11

如果 `list.head()` 和 `list.tail()` 是公有的，可以使用以下这个显式递归版本：

```
<A> Option<List<A>> sequence(List<Option<A>> list) {
  return list.isEmpty()
    ? some(List.list())
    : list.head()
      .flatMap(hh -> sequence(list.tail()).map(x -> x.cons(hh)));
}
```

但是 `list.head()` 和 `list.tail()` 应该只能在 `List` 类中使用，因为这些方法可能会抛出异常。幸运的是，`sequence` 方法也可以通过使用 `foldRight` 和 `map2` 来实现。这样做更好，因为 `foldRight` 使用了基于堆的递归。

```
<A> Option<List<A>> sequence(List<Option<A>> list) {
  return list.foldRight(some(List.list()),
    x -> y -> map2(x, y, a -> b -> b.cons(a)));
}
```

思考以下例子：

```
Function<Integer, Function<String, Integer>> parseWithRadix =
  radix -> string -> Integer.parseInt(string, radix);
Function<String, Option<Integer>> parse16 =
  Option.hlift(parseWithRadix.apply(16));
List<String> list = List.list("4", "5", "6", "7", "8", "9");
Option<List<Integer>> result = Option.sequence(list.map(parse16));
```

这生成了预期的结果，但是效率有点低，因为 `map` 方法和 `sequence` 方法都将调用 `foldRight`。

练习 6.12

定义一个生成相同结果但仅调用一次 `foldRight` 的 `traverse` 方法。这是它的签名：

```
Option<List<B>> traverse(List<A> list, Function<A, Option<B>> f)
```

提示

你需要调用 `traverse` 实现 `sequence`。不要使用递归。首选为你抽象了递归的 `foldRight` 方法。

答案 6.12

首先定义 `traverse` 方法：

```
<A, B> Option<List<B>> traverse(List<A> list,
                                Function<A, Option<B>> f) {
  return list.foldRight(some(List.list()),
                        x -> y -> map2(f.apply(x), y, a -> b -> b.cons(a)));
}
```

然后你可以调用 `traverse` 来重新定义 `sequence` 方法：

```
<A> Option<List<A>> sequence(List<Option<A>> list) {
  return traverse(list, x -> x);
}
```

6.4 Option的其他实用程序

为了使 `Option` 尽量有用，你需要增加一些实用方法。这些方法中有一部分是必需的，而另一些还存有疑问，因为它们违背了函数式编程的精神。然而你必须考虑加上它们。你可能需要一种方法来检查一个 `Option` 到底是 `None` 还是 `Some`。你可能还需要一个 `equals` 方法来比较它们，在这种情况下你可不能忘记定义一个兼容的 `hashCode` 方法。

6.4.1 检查是 `Some` 还是 `None`

迄今为止，你还不需要检查一个 `Option` 到底是一个 `Some` 还是一个 `None`。在理想情况下，你永远不必这样做。虽然在实践中，有时候用这个技巧比用真正的函数式技术更简单。

例如，你将 `map2` 方法定义为

```
<A, B, C> Option<C> map2(Option<A> a,
                        Option<B> b,
                        Function<A, Function<B, C>> f) {
  return a.flatMap(ax -> b.map(bx -> f.apply(ax).apply(bx)));
}
```

这种做法非常聪明，你可能倾向于这个解决方案，因为你想要看起来聪明。但有些人可能会发现以下版本更容易理解：

```
<A, B, C> Option<C> map2(Option<A> a,
```



```

        Option<B> b,
        Function<A, Function<B, C>> f) {
return a.isSome() && b.isSome()
    ? some(f.apply(a.get()).apply(b.getOrThrow()))
    : none();
}

```

测试代码 如果想测试这段代码，你必须首先定义 `isSome` 方法，但这并不是鼓励你使用这种非函数式的技术。你应该总是倾向于第一种形式，但也应该完全理解这两种形式之间的关系。再者，你可能会发现自己有一天需要 `isSome` 方法。

6.4.2 equals 和 hashCode

更重要的是 `equals` 和 `hashCode` 方法的定义。如你所知，这些方法是强相关的，必须一起定义。如果 `Option` 的两个实例的 `equals` 都为 `true`，那么它们的 `hashCode` 方法应该返回相同的值。（反过来不正确，具有相同 `hashCode` 的对象可不总是相等的）。

下面是 `Some` 的 `equals` 和 `hashCode` 的实现：

```

@Override
public boolean equals(Object o) {
    return (this == o || o instanceof Some)
        && this.value.equals(((Some<?>) o).value);
}

@Override
public int hashCode() {
    return Objects.hashCode(value);
}

```

这是与之对应的 `None` 的实现：

```

@Override
public boolean equals(Object o) {
    return this == o || o instanceof None;
}

@Override
public int hashCode() {
    return 0;
}

```

6.5 如何及何时使用Option

你可能知道，Java 8 中引入了一个 `Optional` 类，可能有些人会认为它与你的 `Option` 相同，虽然它的实现方式完全不同，并且缺少了你已添加到 `Option` 中的大部分函数式方法。关于 Java 8 的新特性是否是向函数式编程的转变，存在很多争议。官方的立场是，`Optional` 并不是一个函数式特性。

以下是 Oracle 的 Java 语言架构师 Brian Goetz 在 Stack Overflow 上有关这个话题的回答。问题是“Java 8 的 `getter` 应该返回 `Optional` 类型吗？”以下是 Brian Goetz 的回答：¹

当然，人们想要什么便会做什么。但我们在添加这个特性时确实有明确的意图，它不是一个通用的 `Maybe` 或 `Some` 类型，尽管许多人都想让我们这样做。我们的本意是为库方法的返回类型提供一个有限的机制，需要一个明确的方式来表示“无结果”，而为此使用 `null` 很可能会导致错误。

例如，你不应该用它作为本应是数组或列表的返回结果，而应该返回一个空数组或列表。你基本上不应该把它用作一个字段或方法参数这样的东西。

我认为常规用它作为 `getter` 的返回值肯定会导致其被过度使用。

`Optional` 本身并没有什么应该避免的错误，它只是不像许多人所希望的那样，因此我们非常担心它被过度热情使用的风险。

（公共服务公告：绝不要调用 `Optional.get`，除非你可以证明它不会为 `null`；而要使用一个安全的方法，如 `orElse` 或 `ifPresent`。回想起来，我们应该把 `get` 称为类似 `getOrElseThrowNoSuchElementException` 或能表达出更明确的意图的什么东西：这是一个非常危险的方法，无形中首先破坏了 `Optional` 的本意。教训啊。）

这是一个值得反思的重要答案。首先，这可能是最重要的部分，“人们要什么便会做什么。”没有什么可以加在这里。要什么便做什么。这并不意味着你应该不假思索就做任何想要的东西。但是请随意尝试想到的每一个解决办法。你不应该以

¹ 完整的讨论可以在<http://mng.bz/Rkk1>上阅读。

特定的方式限制使用 `Optional`，只是因为它并不是为这种使用方式而生的。想象一下，曾经想过抓住一块石头用力砸东西的第一个人。他有两个 `option`（双关语！）：避免这样做，因为石头显然不是为像锤子般使用而生的，或者只是试试。

其次，Goetz 说不应该调用 `get`，除非你能证明它永远不会为 `null`。这样做会完全破坏使用 `Option` 的任何收益。但你不需要给 `get` 一个那么长的名字。`getOrThrow` 就能做好这个工作。请注意，返回空列表以指示结果不存在本身并不能解决问题。忘记检查列表是否为空将生成一个 `IndexOutOfBoundsException` 而不是一个 `NullPointerException`。好不到哪儿去！

何时使用 `getOrThrow`

正确的建议是尽量避免使用 `getOrThrow`。作为一个经验法则，每当发现自己在 `Option` 类之外使用这个方法时，你就应该考虑是否还有其他办法。使用 `getOrThrow` 等于放弃了 `Option` 类的函数式安全性。

`List` 类的 `head` 和 `tail` 方法也是如此。如果可能，不应该在 `List` 类之外使用这些方法。如果它正好是 `None` 或 `Nil` 子类的话，直接访问诸如 `List` 或 `Option` 类中包含的值总是会带来 `NullPointerException` 的风险。它也许不可能在库类中避免，但应该在业务类中避免。这就是为什么最好的解决方案是让这个方法成为 `protected`，以使它只能在 `Option` 类内部被调用。

但最重要的还是原来的问题：`getter` 应该返回 `Option`（或 `Optional`）吗？一般来说不应该，因为属性应该为 `final` 并在声明或构造函数中初始化，所以绝对没必要让 `getter` 返回 `Option`。（尽管如此，我必须承认，在构造函数中初始化字段并不能保证在初始化完成前属性不被访问，这个问题不足为惧，如果允许的话，可以很容易通过使类成为 `final` 来解决）。

但是有些属性可能是可选的。例如，人总是会有姓名，但他们可能没有电子邮件。这样要如何表示呢？通过将属性存储为 `Option`。在这种情况下，`getter` 将不得不返回一个 `Option`。说“常规用它作为 `getter` 的返回值肯定会导致其被过度使用”就像在说没有值的属性应该设置为 `null`，并且相应的 `getter` 应该返回 `null`。这完全毁掉了 `Option` 所带来的收益。

那么把 `Option` 作为方法的参数如何？一般来说，不应该这样。你不应该为了复合返回 `Option` 的方法而使用以 `Option` 为参数的方法。例如，要复合以下三个方法，你不需要修改方法使其以 `Option` 为参数：

```
Option<String> getName () {  
    ...  
}  
  
Option<String> validate(String name) {  
    ...  
}  
  
Option<Toon> getToon(String name) {  
    ...  
}
```

鉴于 `validate` 方法是 `Validate` 类的静态方法，`toonMap` 是一个 `Map` 实例并具有 `get` 实例方法，复合这些方法的函数式方法如下：

```
Option<Toon> toon = getName()  
    .flatMap(Validate::validate)  
    .flatMap(toonMap::get)
```

因此，在业务代码中以 `Option` 为参数的方法劳而无功。

还有一个 `Option`（或 `Optional`）应该很少（如果曾经）被使用的原因。一般来说，缺失数据是错误导致的结果，你一般应该通过在命令式 `Java` 中抛出异常的方式来处理。正如我先前所说的，返回 `Option.None` 而不抛出异常就像是捕获了异常并悄悄地吞下了它。通常它不是一个十亿美元的错误，但仍然还是挺大的。你将在第 7 章学习如何处理这种情况。从那以后，几乎不再需要 `Option` 数据类型了。但不要担心，你在本章中学到的所有内容仍然是非常有用的。

`Option` 类型是你多次使用的一种最简单形式的数据类型。它是一个参数化类型，有一个从 `A` 到 `Option<A>` 的方法，有一个可以用来复合 `Option` 实例的 `flatMap` 方法。虽然它本身并不是非常有用，但它已经让你了解到了函数式编程中非常基本的概念。

6.6 总结

- 需要一种方式来表示可选数据，即数据可能存在也可能不存在。
- 空指针是表示数据不存在的最不明智和危险的方法。

- 标记值和空列表是表示数据缺失的其他可行方式，但它们不能很好地复合。
- `Option` 数据类型是表示可选数据的更好方式。`Some` 子类型表示数据存在，`None` 子类型表示数据缺失。
- 函数可以通过将 `map` 和 `flatMap` 方法应用于 `Option` 来允许轻松地复合 `Option`。
- 操作值的函数可以提升为操作 `Option` 实例。
- `List` 可以与 `Option` 复合。可以使用 `sequence` 方法将 `List<Option>` 转换为 `Option<List>`。
- 可以比较 `Option` 实例是否相等。如果包装的值相等，`Some` 子类型的实例就是相等的。由于只有一个 `None` 实例，因而所有 `None` 的实例都是相等的。
- 虽然 `Option` 可能表示计算发生异常的结果，但是与异常有关的所有信息都会丢失。在第7章中，你将学习如何处理这个问题。

7 处理错误和异常

本章要点

- 通过 Either 类型保存关于错误的信息
- 通过有所侧重的 Result 类型更容易地处理错误
- 访问 Result 中的数据
- 应用作用于 Result 数据
- 为操作 Result 提升函数

在第 6 章中，你学习了如何处理可选数据，而无须用 Option 数据类型来处理 null 引用。如你所见，当结果并非源自错误时，这种数据类型非常适合处理数据的缺失。但它并不是一种处理错误的有效方式，这是因为虽然它允许你干净利落地报告数据缺失，但它吞没了缺失的原因。因此，所有缺失的数据都会以相同的方式被处理，并且全得靠调用者来试着找出发生了什么，而一般这是不可能的。

7.1 待解决的问题

大多数时候，数据缺失都是由错误导致的，不是输入数据的问题就是计算的问题。这两种情况很不一样，但它们的结果相同：数据缺失，而数据应该存在才对。

在经典的命令式编程中，当一个函数或方法接收一个对象参数时，大多数程序员知道他们应该测试这个参数是否为 `null`。可要是参数真为 `null`，他们应该做些什么经常是不明确的。回忆第6章中清单6.3所示的示例：

```
Option<String> goofy = toons.get("Goofy").flatMap(Toon::getEmail);

System.out.println(goofy.getOrElse(() -> "No data"));
```

在这个例子中，因为 "Goofy" 键不在 `map` 中，只输出了 "No data"。可以认为这是一个正常情况。但是看看下面这个：

```
Option<String> toon = getName()
    .flatMap(toons::get)
    .flatMap(Toon::getEmail);

System.out.println(toon.getOrElse(() -> "No data"));

Option<String> getName() {
    String name = // 从用户界面获取名称
    return name;
}
```

如果用户输入一个空字符串，你该怎么办？一个明显的方案是验证输入并返回一个 `Option<String>`。如果字符串不合法，你可以返回 `None`。但是，即便你还没有学会如何以函数式的方式让用户输入一个字符串，你也可以确定这样的操作可能会抛出异常。程序将如下所示：

```
Option<String> toon = getName()
    .flatMap(Example::validate)
    .flatMap(toons::get)
    .flatMap(Toon::getEmail);

System.out.println(toon.getOrElse(() -> "No data"));

Option<String> getName() {
    try {
        String name = // 从用户界面获取名称
        return Option.some(name);
    } catch (Exception e) {
        return Option.none();
    }
}

Option<String> validate(String name) {
    return name.length() > 0 ? Option.some(name) : Option.none();
}
```

现在想想可能会发生什么：

- 一切顺利，于是你得到了一个电子邮件并打印到控制台。
- 抛出一个 `IOException`，你只得到了打印到控制台上的 “No data”。
- 用户输入的名称验证失败，你只得到了 “No data”。
- 名称验证通过，但在 `map` 中找不到，你只得到了 “No data”。
- 名称在 `map` 中找到了，但与之相对应的 `toon` 没有电子邮件，你只得到了 “No data”。

你需要控制台上打印出的是不同的消息，以表明在每种情况下究竟都发生了什么。

如果想用已知的类型，你可以用 `Tuple<Option<T>, Option<String>>` 作为每个方法的返回类型，但是这有点儿复杂了。`Tuple` 是一个积类型 (product type)，意味着一个 `Tuple<T, U>` 可以表示的元素数量是 `T` 的可能数量乘以 `U` 的可能数量。你不需要那样，因为每当 `T` 有值时，`U` 就是 `None`。同样的，每当 `U` 为 `Some` 时，`T` 就会是 `None`。你需要的是一个和类型 (sum type)，意味着一个 `E<T, U>` 类型将持有一个 `T` 或一个 `U`，而不是一个 `T` 和一个 `U`。

7.2 Either类型

设计可以持有 `T` 或 `U` 的类型很容易。你只需稍微修改一下 `Option` 类型，通过改变 `None` 类型使其持有一个值即可。你还要更改名称。`Either` 类型有两个私有子类，将它们称为 `Left` 和 `Right`。

清单 7.1 Either 类型

```
public abstract class Either<T, U> {
    private static class Left<T, U> extends Either<T, U> {
        private final T value;

        private Left(T value) {
            this.value = value;
        }

        @Override
```



```

    public String toString() {
        return String.format("Left(%s)", value);
    }
}

private static class Right<T, U> extends Either<T, U> {

    private final U value;

    private Right(U value) {
        this.value = value;
    }

    @Override
    public String toString() {
        return String.format("Right(%s)", value);
    }
}

public static <T, U> Either<T, U> left(T value) {
    return new Left<>(value);
}

public static <T, U> Either<T, U> right(U value) {
    return new Right<>(value);
}
}

```

现在你可以轻松使用 `Either` 而不是 `Option` 来表示可能由于错误而缺失的值。你必须用数据的类型和错误的类型来参数化 `Either`。依照惯例，用 `Right` 子类来表示成功（“right”即“正确”），用 `Left` 来表示错误。但是你不会将子类称为 `Wrong`，因为 `Either` 类型也可以用于表示两种不同类型的合法数据。

当然，你必须选择什么类型将代表错误。你可以选择用 `String` 表示错误消息，也可以选择用 `Exception`。例如，在第6章中定义的 `max` 函数可以进行如下修改：

```

<A extends Comparable<A>> Function<List<A>, Either<String, A>> max() {
    return xs -> xs.isEmpty()
        ? Either.left("max called on an empty list")
        : Either.right(xs.foldLeft(xs.head(), x -> y -> x.compareTo(y) < 0 ?
                                x : y));
}

```

7.2.1 复合 Either

要复合返回 `Either` 的方法或函数，你同样需要定义在 `Option` 类中已定义过

的方法。

练习 7.1

定义一个 `map` 方法，接收一个从 `A` 到 `B` 的函数，并将 `Either<E, A>` 转换为 `Either<E, B>`。`map` 方法的签名如下：

```
public abstract <B> Either<E, B> map(Function<A, B> f);
```

提示

我使用类型参数 `E` 和 `A` 来明确映射的对应关系，`E` 代表错误（error）。但是其实可以定义两个 `map` 方法（称它们为 `mapLeft` 和 `mapRight`）以映射 `Either` 实例的某一侧。换句话说，你正在开发一个只能映射一侧的“有所侧重”版的 `Either`。

答案 7.1

`Left` 的实现比 `Option` 的 `None` 实现稍微复杂一些，因为你必须构造一个全新的 `Either` 来持有相同（错误）的原始值：

```
public <B> Either<E, B> map(Function<A, B> f) {
    return new Left<>(value);
}
```

`Right` 的实现与 `Some` 几乎完全相同：

```
public <B> Either<E, B> map(Function<A, B> f) {
    return new Right<>(f.apply(value));
}
```

练习 7.2

定义 `flatMap` 方法，给定从 `A` 到 `Either<E, B>` 的函数，并将 `Either<E, A>` 转换为 `Either<E, B>`。`flatMap` 方法的签名如下：

```
public abstract <B> Either<E, B> flatMap(Function<A, Either<E, B>> f);
```

答案 7.2

`Left` 的实现与 `map` 方法完全相同：

```
public <B> Either<E, B> flatMap(Function<A, Either<E, B>> f) {
    return new Left<>(value);
}
```

Right 的实现与 Option.flatMap 方法相同：

```
public <B> Either<E, B> flatMap(Function<A, Either<E, B>> f) {
    return f.apply(value);
}
```

练习 7.3

使用以下签名定义方法 getOrElse 和 orElse：

```
A getOrElse(Supplier<A> defaultValue)
Either<E, A> orElse(Supplier<Either<E, A>> defaultValue)
```

提示

不是所有的练习都有令人满意的答案！

答案 7.3

orElse 方法可以定义在 Either 类中，因为同一个实现适用于两个子类：

```
public Either<E, A> orElse(Supplier<Either<E, A>> defaultValue) {
    return map(x -> this).getOrElse(defaultValue);
}
```

getOrElse 方法的解决方案很直观。在 Right 子类中，你只需返回包含的值：

```
public A getOrElse(Supplier<A> defaultValue) {
    return value;
}
```

在 Left 子类中，只需返回默认值：

```
public A getOrElse(Supplier<A> defaultValue) {
    return defaultValue.get();
}
```

这种方法可以工作，但它离完美还相距甚远。问题是如果没有值，你根本就无从知道发生了什么。你只是得到了默认值，甚至都不知道它是计算的结果还是错误的结果。要正确处理错误情况，你需要一个有所侧重版的 Either，其中 left 类型是已知的。你可以使用一个已知的固定类型为 Left 类创建一个专门的版本，而不是使用 Either（顺便提一句，它有许多其他有趣的用途）。

你可能会问的第一个问题是，“我应该使用什么类型？”显然脑海中会冒出两

种不同的类型：String 和 RuntimeException。字符串可以像异常那样持有错误消息，但许多错误情况都会生成异常。用 String 作为 Left 的值类型将强制你忽略异常中的有关信息，并仅使用其包含的消息。因此最好使用 RuntimeException 作为 Left 值。通过这种方式，如果只有一条消息，你也可以把它包装成一个异常。

7.3 Result类型

由于新类型通常表示可能失败的计算结果，所以称其为 Result。它与 Option 类型非常相似，区别在于其子类被命名为 Success 和 Failure，如清单 7.2 所示。

清单 7.2 Result 类

```
import java.io.Serializable;

public abstract class Result<V> implements Serializable {

    private Result() {
    }

    private static class Failure<V> extends Result<V> {

        private final RuntimeException exception;

        private Failure(String message) {
            super();
            this.exception = new IllegalStateException(message);
        }

        private Failure(RuntimeException e) {
            super();
            this.exception = e;
        }

        private Failure(Exception e) {
            super();
            this.exception = new IllegalStateException(e.getMessage(), e);
        }

        @Override
        public String toString() {
            return String.format("Failure(%s)", exception.getMessage());
        }
    }
}
```

Result 类只接收一个对应于成功值的类型参数

Failure 子类包含一个 RuntimeException

构造函数是私有的。如果 Failure 是通过一条消息构造的，它就会被包装到一个 RuntimeException 中（更具体地说，是 IllegalStateException 子类）

如果通过 RuntimeException 构造，它将按原样存储

如果通过可控异常构造，它会被包装到一个 RuntimeException 里

```

private static class Success<V> extends Result<V> {
    private final V value;
    private Success(V value) {
        super();
        this.value = value;
    }
    @Override
    public String toString() {
        return String.format("Success(%s)", value.toString());
    }
}

public static <V> Result<V> failure(String message) {
    return new Failure<>(message);
}

public static <V> Result<V> failure(Exception e) {
    return new Failure<V>(e);
}

public static <V> Result<V> failure(RuntimeException e) {
    return new Failure<V>(e);
}

public static <V> Result<V> success(V value) {
    return new Success<>(value);
}

```

Success 子类
存储成功的值

使用工厂
方法来创
建 Result
实例

这个类很像附带了存储异常的 Option 类。

7.3.1 为 Result 类添加方法

Result 类需要你在 Option 和 Either 类中定义过的相同方法，但略有不同。

练习 7.4

为 Result 类定义 map、flatMap、getOrElse 和 orElse。对 getOrElse 而言，你可以定义两个方法：其一接收一个值为参数，另一个接收一个 Supplier。签名如下：

```

public abstract V getOrElse(final V defaultValue);
public abstract V getOrElse(final Supplier<V> defaultValue);
public abstract <U> Result<U> map(Function<V, U> f);
public abstract <U> Result<U> flatMap(Function<V, Result<U>> f);
public Result<V> orElse(Supplier<Result<V>> defaultValue);

```

当默认值为字面量时，`getOrElse` 的第一个版本很有用，因为它已经被赋值了。在这种情况下，你无须使用惰性求值。

答案 7.4

这次你将顺利搞定 `getOrElse`，因为只需抛出包含在 `Failure` 中的异常即可。所有其他方法都非常类似于 `Either` 类。以下是 `Success` 类的实现：

```
public V getOrElse(V defaultValue) {
    return value;
}

public V getOrElse(Supplier<V> defaultValue) {
    return value;
}

public <U> Result<U> map(Function<V, U> f) {
    try {
        return success(f.apply(successValue()));
    } catch (Exception e) {
        return failure(e.getMessage(), e);
    }
}

public <U> Result<U> flatMap(Function<V, Result<U>> f) {
    try {
        return f.apply(successValue());
    } catch (Exception e) {
        return failure(e.getMessage());
    }
}
```

以下是 `Failure` 类的实现：

```
public V getOrElse(V defaultValue) {
    return defaultValue;
}

public V getOrElse(Supplier<V> defaultValue) {
    return defaultValue.get();
}

public <U> Result<U> map(Function<V, U> f) {
    return failure(exception);
}

public <U> Result<U> flatMap(Function<V, Result<U>> f) {
    return failure(exception);
}
```

正如 Option，由于类型无效，map 和 flatMap 不能在 Failure 类中返回 this。

最后，你可以在父类中定义 orElse 方法，因为其实现对两个子类都有效：

```
public Result<V> orElse(Supplier<Result<V>> defaultValue) {
    return map(x -> this).getOrElse(defaultValue);
}
```

7.4 Result模式

现在能够以函数式的方式使用 Result 类了，这就意味着可以通过复合方法来表示可能成功或失败的计算。这非常重要，因为 Result 和类似的类型通常被描述为可能包含也可能不包含值的容器。这个描述是片面的。Result 是一个值的计算上下文，这个值可能存在也可能不存在。并不通过获取值来使用它，而是通过用其特定的方法来复合 Result 的实例。

例如，你可以修改先前的 ToonMail 示例来使用此类。首先，你必须修改 Map 和 Toon 类，如清单 7.3 和清单 7.4 所示。

清单 7.3 修改了 Map 类的 get 方法返回一个 Result

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;

public class Map<T, U> {

    private final ConcurrentMap<T, U> map = new ConcurrentHashMap<>();

    public static <T, U> Map<T, U> empty() {
        return new Map<>();
    }

    public static <T, U> Map<T, U> add(Map<T, U> m, T t, U u) {
        m.map.put(t, u);
        return m;
    }

    public Result<U> get(final T t) {
        return this.map.containsKey(t)
            ? Result.success(this.map.get(t))
            : Result.failure(String.format("Key %s not found in map", t));

        // 如果映射中包含了键，则返回一个 Success，包含了获取到的对象
        // 否则，返回一个 Failure，包含了错误消息
    }

    public Map<T, U> put(T t, U u) {
```

```

    return add(this, t, u);
}

public Map<T, U> removeKey(T t) {
    this.map.remove(t);
    return this;
}
}

```

清单 7.4 修改了 mail 属性的 Toon 类

```

public class Toon {

    private final String firstName;
    private final String lastName;
    private final Result<String> email;

    Toon(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = Result.failure(String.format("%s %s has no mail",
                                                    firstName, lastName));
    }

    Toon(String firstName, String lastName, String email) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = Result.success(email);
    }

    public Result<String> getEmail() {
        return email;
    }
}

```

如果没有提供邮件，则存储一个 Failure

如果对象是通过 email 构造的，则将其包装在一个 Success 里

getEmail 方法返回一个 Result（不是 Success 就是 Failure）

现在你可以如清单 7.5 所示修改 ToonMail 程序。

清单 7.5 用 Result 修改后的程序

```

import java.io.IOException;

public class ToonMail {

    public static void main(String[] args) {
        Map<String, Toon> toons = new Map<String, Toon>()
            .put("Mickey", new Toon("Mickey", "Mouse", "mickey@disney.com"))
            .put("Minnie", new Toon("Minnie", "Mouse"))
            .put("Donald", new Toon("Donald", "Duck", "donald@disney.com"));
        Result<String> result =
            getName().flatMap(toons::get).flatMap(Toon::getEmail);
    }
}

```

通过 flatMap 复合了返回 Result 的方法


```

        System.out.println(result);
    }

    public static Result<String> getName() {
        return Result.success("Mickey");
    }
}

```

getName 方法模拟了一个输入，可能导致一个 Failure

清单 7.5 中的程序使用 `getName` 方法来模拟可能会抛出异常的输入操作。要表示抛出的异常，你只需返回一个包装了这个异常的 `Failure`。

请注意返回 `Result` 的各种操作是如何复合的。你无须访问 `Result` 中所包含的值（它可能是一个异常）。`flatMap` 方法可以用于这样的复合。

尝试用 `getName` 方法的各种实现来运行这段程序，例如：

```

return Result.success("Mickey");
return Result.failure(new IOException("Input error"));
return Result.success("Minnie");
return Result.success("Goofy");

```

以下是程序在每种情况下所打印的内容：

```

Success(mickey@disney.com)
Failure(Input error)
Failure(Minnie Mouse has no mail)
Failure(Key Goofy not found in map)

```

这个结果看起来还算不错，但其实并非如此。问题是没有电子邮件的 `Minnie` 和不在 `map` 中的 `Goofy` 都被报告为失败。它们也许算是失败，但它们也可能是正常情况。毕竟，如果没有电子邮件就算是一个失败的话，你不应该允许创建一个没有电子邮件的 `Toon` 实例。很显然这不是一个失败，只是可选数据而已。对于 `map` 也是如此。如果键不在 `map` 中（假定它应该在那里），那么可能算是一个错误，但是在 `map` 的角度上看，它只是可选数据而已。

你可能会认为这不是一个问题，因为你为此已经有了一个类型：在第 6 章中开发的 `Option` 类型。但是看看你复合函数的方式：

```

getName().flatMap(toons::get).flatMap(Toon::getEmail);

```

这么做可行仅仅是因为 `getName`、`Map.get` 和 `Toon.getEmail` 都返回了一个 `Result`。如果 `Map.get` 和 `Toon.getEmail` 返回 `Option`，它们就不再能够复合 `getName`。

仍然可以将 Result 与 Option 相互转换。例如，你可以在 Result 中添加一个 toOption 方法：

```
public abstract Option<V> toOption()
```

Success 的实现是

```
public Option<V> toOption() {  
    return Option.some(value);  
}
```

Failure 的实现是

```
public Option<V> toOption() {  
    return Option.none();  
}
```

接下来你就可以如下使用：

```
Option<String> result =  
    getName().toOption().flatMap(toons::get).flatMap(Toon::getEmail);
```

当然，这将要求你使用第 6 章（清单 6.2）中定义的 Map 版本和指定版本的 Toon 类：

```
public class Toon {  
    private final String firstName;  
    private final String lastName;  
    private final Option<String> email;  
  
    Toon(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.email = Option.none();  
    }  
  
    Toon(String firstName, String lastName, String email) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.email = Option.some(email);  
    }  
  
    public Option<String> getEmail() {  
        return email;  
    }  
}
```

但是你会失去使用 Result 的所有好处！如果现在 getName 方法中抛出一个

异常，它仍然包含在一个 `Failure` 中，但是这个异常在 `toOption` 方法中丢失了，程序只会打印

```
none
```

你也许会认为应该换一种方式，把一个 `Option` 转换为 `Result`。这可以工作(虽然在你的例子中，应该在由 `Map.get` 和 `Toon.getMail` 返回的两个 `Option` 实例上调用新的 `toResult` 方法)，但这很没意思，因为你经常需要将 `Option` 转换为 `Result`，一个更好的办法是将这个转换置于 `Result` 类中。你需要做的全部工作就是创建一个与 `None` 相对应的新子类，因为 `Some` 不需要转换，只需将其名称更改为 `Success` 即可。清单 7.6 展示了新的 `Result` 类，它有被称为 `Empty` 的新子类。

清单 7.6 处理错误和可选数据的全新 `Result` 类

```
public abstract class Result<V> implements Serializable {

    @SuppressWarnings("rawtypes")
    private static Result empty = new Empty();

    ...

    private static class Empty<V> extends Result<V> {

        public Empty() {
            super();
        }

        @Override
        public V getOrElse(final V defaultValue) {
            return defaultValue;
        }

        @Override
        public <U> Result<U> map(Function<V, U> f) {
            return empty();
        }

        @Override
        public <U> Result<U> flatMap(Function<V, Result<U>> f) {
            return empty();
        }

        @Override
        public String toString() {
            return "Empty()";
        }
    }
}
```

← 如同 `Option` 中的 `None` 实例，`Result` 包含了一个单例 `Empty`，它是一个原始类型 (raw type)

```

@Override
public V getOrElse(Supplier<V> defaultValue) {
    return defaultValue.get();
}
}

private static class Failure<V> extends Empty<V> {

    private final RuntimeException exception;

    private Failure(String message) {
        super();
        this.exception = new IllegalStateException(message);
    }

    private Failure(RuntimeException e) {
        super();
        this.exception = e;
    }

    private Failure(Exception e) {
        super();
        this.exception = new IllegalStateException(e.getMessage(), e);
    }

    @Override
    public String toString() {
        return String.format("Failure(%s)", exception.getMessage());
    }

    @Override
    public <U> Result<U> map(Function<V, U> f) {
        return failure(exception);
    }

    @Override
    public <U> Result<U> flatMap(Function<V, Result<U>> f) {
        return failure(exception);
    }
}

...

@SuppressWarnings("unchecked")
public static <V> Result<V> empty() {
    return empty;
}
}

```

Failure 类继承自 Empty 类，便不必重新定义具有相同实现的 getOrElse 和 OrElse 方法

Failure 类重写了 Empty 的 map 和 flatMap 方法，以便使用包含的异常

如同 Option 中的 none 方法，empty 方法返回 Empty 单例

现在可以再次修改你的 ToonMail 程序，如清单 7.7 到清单 7.9 所示。

清单 7.7 Map 类使用新的 Result.Empty 类作为可选数据

```

public class Map<T, U> {
    private final ConcurrentMap<T, U> map = new ConcurrentHashMap<>();

    public static <T, U> Map<T, U> empty() {
        return new Map<>();
    }

    public static <T, U> Map<T, U> add(Map<T, U> m, T t, U u) {
        m.map.put(t, u);
        return m;
    }

    public Result<U> get(final T t) {
        return this.map.containsKey(t)
            ? Result.success(this.map.get(t))
            : Result.empty();
    }

    public Map<T, U> put(T t, U u) {
        return add(this, t, u);
    }

    public Map<T, U> removeKey(T t) {
        this.map.remove(t);
        return this;
    }
}

```

如果在 map 中找不到这个键，get 方法现在会返回 Result.empty()

清单 7.8 Toon 类使用 Result.Empty 作为可选数据

```

public class Toon {
    private final String firstName;
    private final String lastName;
    private final Result<String> email;

    Toon(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = Result.empty();
    }

    Toon(String firstName, String lastName, String email) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = Result.success(email);
    }
}

```

如果构造不含电子邮件的实例，那么属性将被设为 Result.empty()

```

public Result<String> getEmail() {
    return email;
}
}

```

清单 7.9 ToonMail 程序正确处理可选数据

```

public class ToonMail {

    public static void main(String[] args) {
        Map<String, Toon> toons = new Map<String, Toon>()
            .put("Mickey", new Toon("Mickey", "Mouse", "mickey@disney.com"))
            .put("Minnie", new Toon("Minnie", "Mouse"))
            .put("Donald", new Toon("Donald", "Duck", "donald@disney.com"));
        Result<String> result =
            getName().flatMap(toons::get).flatMap(Toon::getEmail);
        System.out.println(result);
    }

    public static Result<String> getName() {
        return Result.success("Mickey");
        //return Result.failure(new IOException("Input error"));
        //return Result.success("Minnie");
        //return Result.success("Goofy");
    }
}

```

通过 flatMap 复合方法，如清单 7.5 所示

测试全部情况的多种实现

现在你的程序可以为 getName 方法的每个实现打印出以下结果（在清单 7.9 中注释掉了）：

```

Success(mickey@disney.com)
Failure(Input error)
Empty()
Empty()

```

你可能认为缺少了某些东西，因为不能区分两种不同情况下的空值，但却并非如此。可选数据不需要错误消息，因此如果你认为需要一条消息，则数据就不是可选的。success 的结果是可选的，但那样的话需要一个消息，因此你应该使用一个 Failure。这样会创建一个异常，但谁也不会逼你抛出它！

7.5 Result处理进阶

迄今为止，你已经看到了 Result 的使用场景非常有限。绝不应该直接访问 Result 所包装的值（如果值存在的话）。你在上一个示例中使用 Result 的方式对

应于更加简单的特定用例组合：获取一个计算的结果，并将其用于下一个计算的输入。还有更具体的使用情况。你可以选择只有匹配一些断言（意即一些条件）时使用 `result`。你还可以使用 `failure`，不过需要将其映射到其他东西，或者将其转换为异常的 `success (Success<Exception>)`。你可能还需要使用几个 `Result` 作为单个计算的输入。你可能会受益于一些从计算中创建 `Result` 的辅助方法，以便处理旧代码。最后，有时你需要对 `Result` 应用作用。

7.5.1 应用断言

你经常需要对 `Result` 应用断言。它很容易被抽象，所以你可以只编写一次。

练习 7.5

编写一个 `filter` 方法，接收一个条件并返回一个 `Result<T>`，条件由从 `T` 到 `Boolean` 函数 `T` 表示。视包装的值是否满足条件而定，`Result` 会是一个 `Success` 或 `Failure`。签名为

```
Result<T> filter(Function<T, Boolean> f);
```

创建第二个方法，以条件为第一个参数，以 `String` 为第二个参数，并在可能的 `Failure` 情况下使用将该字符串。

提示

虽然可以在 `Result` 类中定义抽象方法，并在子类中实现它们，但请尽量不要这样做。而是用先前定义的一个或多个方法在 `Result` 类中创建一个实现。

答案 7.5

你需要创建一个函数，以包装的值为参数，应用该函数并在满足条件时返回相同的 `Result`，否则就返回 `Empty`（或 `Failure`）。然后你所要做的就是对这个函数 `flatMap`：

```
public Result<T> filter(Function<T, Boolean> p) {  
    return flatMap(x -> p.apply(x)  
        ? this  
        : failure("Condition not matched"));  
}
```

```
public Result<T> filter(Function<T, Boolean> p, String message) {  
    return flatMap(x -> p.apply(x)
```

```

    ? this
    : failure(message));
}

```

练习 7.6

定义一个 `exists` 方法，接收从 `T` 到 `Boolean` 的函数，并在包装的值与条件匹配时返回 `true`，否则返回 `false`。这是方法签名：

```
boolean exists(Function<T, Boolean> p);
```

提示

还是尽量不要在每个子类中定义一个实现。相反，在父类中通过可用的方法来创建一个实现。

答案 7.6

答案只是简单地将函数映射到 `Result<T>`，给定一个 `Result<Boolean>`，然后用 `getOrElse` 并传入 `false` 作为默认值。你不必使用 `Supplier`，因为默认值是字面量：

```

public boolean exists(Function<T, Boolean> p) {
    return map(p).getOrElse(false);
}

```

使用 `exists` 作为这个方法的名称似乎是可商榷的。但相同的方法也可以应用于列表，如果至少一个元素满足条件则返回 `true`，所以用相同的名称是有意义的。可能有些人会认为这个实现也可以用于当列表中的所有元素都满足条件时返回 `true` 的 `forall` 方法。你可以选择其他名称，或者在 `Result` 类中定义一个具有相同实现的 `forall` 方法。关键在于理解究竟是什么使 `List` 和 `Result` 相似，又是什么使它们不同。

7.5.2 映射 Failure

有时将 `Failure` 转换为另一种形式会对我们有所帮助，如清单 7.10 所示。

清单 7.10 内存监控器

```

package com.fpinjava.handlererrors.listing07_10;

import com.fpinjava.common.List;

```



```

import com.fpinjava.common.Result;
import javax.management.Notification;
import javax.management.NotificationEmitter;
import javax.management.NotificationListener;
import java.lang.management.ManagementFactory;
import java.lang.management.MemoryNotificationInfo;
import java.lang.management.MemoryPoolMXBean;

public class MemoryMonitor {

    public static void monitorMemory(double threshold) {
        findPSOldGenPool().forEachOrThrow(poolMxBean ->
            poolMxBean.setCollectionUsageThreshold((int) Math.floor(poolMxBean
                .getUsage().getMax() * threshold)));
        NotificationEmitter emitter = (NotificationEmitter) ManagementFactory.
            getMemoryMXBean();
        emitter.addNotificationListener(notificationListener, null, null);
    }

    private static NotificationListener notificationListener =
        (Notification notification, Object handBack) -> {
        if (notification.getType().equals(MemoryNotificationInfo
            .MEMORY_COLLECTION_THRESHOLD_EXCEEDED)) {
            // 干净利落地关闭程序
        }
    };

    private static Result<MemoryPoolMXBean> findPSOldGenPool() {
        return List.fromCollection(ManagementFactory.getMemoryPoolMXBeans())
            .first(x -> x.getName().equals("PS Old Gen"));
    }
}

```

第一个方法返回一个 Result。在错误的情况下，它将是一个包含无用错误消息的 Failure。应该将其替换为有意义的消息

在 Java 多线程程序中，OutOfMemoryError (OOM) 经常使线程崩溃，但并不会使应用程序崩溃，而是使其处于一个不确定的状态。要解决这个问题，你必须捕获错误，并且干净利落地停止应用程序。

通常捕获 OOM 是在 UncaughtExceptionHandler 的帮助下完成的。这种方式允许你将处理程序放在低层次的库中，并继续要求业务开发人员不要捕获 OOM。但是当 OOM 被捕获时，有时并没有足够的内存来运行处理程序，从而导致应用程序行为不稳定。这个问题的一种解决办法是使用 MemoryPoolMXBean 来监视内存。这种办法允许你注册通知处理程序，当没有足够的内存释放时在垃圾回收之后自动调用。

在示例中，如果调用 `monitorMemory` 方法并传入 0.8 为参数，那么在垃圾回收之后要是还有超过 80% 的堆被占用，则将立即调用通知监听器。此时，你希望有足够的内存来清楚地将问题写入日志并停止应用程序。

这个程序工作得挺好（尽管代码很糟糕，主要源于 Java 库的编写方式，方法以 `null` 为参数，并强制你将 `MemoryPoolMXBean` 转换为 `NotificationEmitter`，但这是另一回事了）。

请注意，这个程序使用你还没有在 `List` 上定义的 `first` 方法。这个方法与 `filter` 方法非常相似，虽然它返回的 `Result` 可能包装了满足条件的第一个元素。

尽管程序能够工作，但是有一个问题：如果基于任何原因，`findPSOldGenPool` 方法返回了一个 `Failure`，无论是错误地拼写为“PS Old Gen”还是因为使用了名称改变了的 Java 新版本，你将在 `Failure` 中得到以下错误消息：

```
No element satisfying function com.fpinjava.handlingerrors
    .listing07_10.MemoryMonitor$
$Lambda$3/1096979270@7b23ec81 in list
[sun.management.MemoryPoolImpl@3feba861,
sun.management.MemoryPoolImpl@5b480cf9,
sun.management.MemoryPoolImpl@6f496d9f,
sun.management.MemoryPoolImpl@723279cf,
sun.management.MemoryPoolImpl@10f87f48,
sun.management.MemoryPoolImpl@b4c966a, NIL]
```

练习 7.7

定义一个 `mapFailure` 方法，该方法以一个 `String` 为参数，并以其为错误消息将一个 `Failure` 转换为另一个 `Failure`。如果 `Result` 为 `Empty` 或 `Success`，这个方法应该什么也不做。

提示

在父类中定义抽象方法。

答案 7.7

这是父类中的抽象方法：

```
public abstract Result<T> mapFailure(String s);
```

`Empty` 和 `Success` 实现只返回 `this`：

```
public Result<T> mapFailure(String s) {
```

```
return this;
}
```

Failure 实现将现有异常封装到使用给定消息创建的新异常中。然后通过调用相应的静态工厂方法创建一个新的 Failure:

```
public Result<T> mapFailure(String s) {
    return failure(new IllegalStateException(s, exception));
}
```

你可以选择 RuntimeException 作为异常类型, 或更具体的自定义 RuntimeException 子类型。请注意, 其他一些同类方法可能也会有用, 如以下这些:

```
public abstract Result<T> mapFailure(String s, Exception e);
public abstract Result<T> mapFailure(Exception e);
```

另一个有用的方法是给定一个 String 消息, 将一个 Empty 映射到一个 Failure。

7.5.3 增加工厂方法

你已经了解了如何用 一个值创建 Success 和 Failure。一些其他用例是如此频繁被使用, 值得将它们抽象为新增的静态工厂方法。为了适应旧库, 也许你经常需要用可能为 null 的值来创建 Result。为此, 你可以使用具有以下签名的静态工厂方法:

```
public static <T> Result<T> of(T value)
public static <T> Result<T> of(T value, String message)
```

一个接收从 T 到 Boolean 的函数和一个 T 实例并创建 Result 的方法也可能会有用:

```
public static <T> Result<T> of(Function<T, Boolean> predicate, T value)
public static <T> Result<T> of(Function<T, Boolean> predicate,
    T value, String message)
```

练习 7.8

定义这些静态工厂方法。

提示

你必须选择在每种情况下返回什么。

答案 7.8

这个练习没有什么难度。这是基于当没有使用错误消息时返回 Empty，否则返回 Failure 的可能实现：

```
public static <T> Result<T> of(T value) {
    return value != null
        ? success(value)
        : Result.failure("Null value");
}

public static <T> Result<T> of(T value, String message) {
    return value != null
        ? success(value)
        : failure(message);
}

public static <T> Result<T> of(Function<T, Boolean> predicate, T value) {
    try {
        return predicate.apply(value)
            ? success(value)
            : empty();
    } catch (Exception e) {
        String errorMessage =
            String.format("Exception while evaluating predicate: %s", value);
        return Result.failure(new IllegalStateException(errorMessage, e));
    }
}

public static <T> Result<T> of(Function<T, Boolean> predicate,
    T value, String message) {
    try {
        return predicate.apply(value)
            ? Result.success(value)
            : Result.failure(String.format(message, value));
    } catch (Exception e) {
        String errorMessage =
            String.format("Exception while evaluating predicate: %s",
                String.format(message, value));
        return Result.failure(new IllegalStateException(errorMessage, e));
    }
}
```

请注意，你需要处理 `message` 参数为 `null` 的情况，否则可能抛出一个 NPE，导致一个 `null` 消息被认为是一个 bug。与此相反，你可以检查参数，并在 `null` 的情况下使用默认值。这取决于你。在任何情况下，始终检查 `null` 参数的处理都应该被抽象出来，其原因你将会在第 15 章中看到。

7.5.4 应用作用

迄今为止,你还未对包装在 `Result` 中的值应用任何作用,而只是获取这些值(通过 `getOrNull`)。这可不令人满意,因为它破坏了使用 `Result` 的收益。另一方面,你还没有学会函数式地应用作用的必要技术。作用包括修改外界的任何东西,例如输出到控制台、文件、数据库或可变组件中的字段,以及在本地或通过网络发送消息。

我即将展示的技术不是函数式的,但它是一个有趣的抽象,允许你使用 `Result` 而无须知道涉及的函数式技术。你可以使用这里展示的技术,直到看到函数式版本为止,你甚至会发现这个技术已经强大到可以在日常使用。

注意 本节讨论的技术是 Java 8 的函数式结构所采用的方式,不用诧异,因为 Java 并不是一门函数式编程语言。

使用你在第3章中开发的 `Effect` 接口来应用作用。这是一个非常简单的函数式接口:

```
public interface Effect<T> {  
    void apply(T t);  
}
```

你可以把这个接口命名为 `Consumer` 并改为定义一个 `accept` 方法,就像 Java 8 中一样。我已经说过这个名字是非常糟糕的选择,因为一个 `Consumer` (消费者)应该有一个 `consume` (消费)方法。但事实上, `Consumer` 并不 `consume` 任何东西——在对值应用作用之后,该值保持不变,并且仍然可以用于进一步的计算或作用。

练习 7.9

定义一个 `forEach` 方法,它以一个 `Effect` 为参数,并将其应用于包装值。

提示

在 `Result` 类中定义一个抽象方法并在每个子类中实现。

答案 7.9

以下是 `Result` 中的抽象方法声明:

```
public abstract void forEach(Effect<T> ef)
```

`Empty` 和 `Failure` 的实现什么也不做。因此,你只需在 `Empty` 中实现该方法,

因为 Failure 继承了该类：

```
public void forEach(Effect<T> ef) {  
    // Empty. Do nothing.  
}
```

Success 的实现很直观。你只需将作用应用于值：

```
public void forEach(Effect<T> ef) {  
    ef.apply(value);  
}
```

这个 forEach 方法将完美适用你在第 6 章中创建的 Option 类。但对于 Result 就行不通了。一般来说，你要对 failure 采取特殊措施。处理 failure 的一个简单办法就是抛出异常。

练习 7.10

定义 forEachOrThrow 方法来处理这个用例。这是它在 Result 类中的签名：

```
public abstract void forEachOrThrow(Effect<T> ef)
```

提示

你可以选择 Empty 的实现。

答案 7.10

Success 的实现与 forEach 方法的实现相同。Failure 的实现只是抛出包装的异常：

```
public void forEachOrThrow(Effect<T> ef) {  
    throw exception;  
}
```

Empty 的实现更成问题。你可以选择什么事情也不做，认为 Empty 不是一个错误。你也可以决定调用 forEachOrThrow，也就是说，把数据的缺失转换为一个错误。这是一个非常艰难的决定。Empty 本身不是错误。如果要使它成为一个错误，你可以使用一个 mapFailure 方法，所以在 Empty 中最好让 forEachOrThrow 实现为一个什么也不做的方法。

练习 7.11

将作用应用于 Result 时，常见的用例是对 Success 应用一个作用，而对

Failure 则以某种方式来处理异常。forEachOrThrow 方法可以抛出异常，但有时你只是想记录错误并继续。定义一个 forEachOrException 方法而不是定义记录日志的方法，当存在值时应用一个作用并返回一个 Result。如果原始的 Result 是一个 Success，则 Result 将为 Empty；如果是一个 Failure，则 Result 将为 Empty 或 Success<RuntimeException>。

答案 7.11

该方法在 Result 父类中声明为 abstract：

```
public abstract Result<RuntimeException> forEachOrException(Effect<T> ef)
```

Empty 的实现返回 Empty：

```
public Result<RuntimeException> forEachOrException(Effect<T> ef) {
    return empty();
}
```

Success 的实现将作用应用于包装的值，并返回 Empty：

```
public Result<RuntimeException> forEachOrException(Effect<T> ef) {
    ef.apply(value);
    return empty();
}
```

Failure 的实现返回一个 Success<RuntimeException> 以持有原来的异常，以便你可以处理它：

```
public Result<RuntimeException> forEachOrException(Effect<T> ef) {
    return success(exception);
}
```

这个方法的典型用例如下（使用带有 log 方法的假想 Logger 类型）：

```
Result<Integer> result = getComputation();
result.forEachOrException(System.out::println).forEach(Logger::log);
```

请记住，这些方法不是函数式的，但是它们都是使用 Result 的好办法。如果想要函数式地应用作用，那你得等到第 13 章。

7.5.5 Result 复合进阶

Result 的用法与 Option 大致相同。在上一章中，你定义了用于复合 Option

的 lift 方法, 通过将从 A 到 B 的函数转换为从 Option<A> 到 Option 的函数。可以对 Result 同样再来一遍。

练习 7.12

为 Result 编写 lift 方法。它是一个在 Result 类中具有以下签名的静态方法:

```
static <A, B> Function<Result<A>, Result<B>>> lift(final Function<A, B> f)
```

答案 7.12

这个答案非常简单:

```
public static <A, B> Function<Result<A>, Result<B>>> lift(final Function<A, B> f) {
    return x -> {
        try {
            return x.map(f);
        } catch (Exception e) {
            return failure(e);
        }
    };
}
```

练习 7.13

定义 lift2 用于提升从 A 到 B 到 C 的函数, lift3 用于从 A 到 B 到 C 到 D 的函数, 签名如下:

```
public static <A, B, C> Function<Result<A>, Function<Result<B>, Result<C>>>> lift2(Function<A, Function<B, C>>> f)
public static <A, B, C, D> Function<Result<A>, Function<Result<B>, Function<Result<C>, Result<D>>>>> lift3(Function<A, Function<B, Function<C, D>>>> f)
```

答案 7.13

答案如下:

```
public static <A, B, C> Function<Result<A>, Function<Result<B>, Result<C>>>> lift2(Function<A, Function<B, C>>> f) {
    return a -> b -> a.map(f).flatMap(b::map);
}
public static <A, B, C, D> Function<Result<A>, Function<Result<B>, Function<Result<C>, Result<D>>>>> lift3(Function<A, Function<B, Function<C, D>>>> f) {
    return a -> b -> c -> a.map(f).flatMap(b::map).flatMap(c::map);
}
```


我猜你能够发现规律了。可以同样为任意数量的参数定义 lift。

练习 7.14

在第 6 章中，你定义了一个 map2 方法，接收一个 Option<A>，一个 Option，以及一个从 A 到 B 到 C 的函数为参数，并返回一个 Option<C>。为 Result 定义一个 map2 方法。

提示

不要用你为 Option 定义的方法，应该使用 lift2 方法。

答案 7.14

为 Option 定义的解决办法是

```
<A, B, C> Option<C> map2(Option<A> a,
                          Option<B> b,
                          Function<A, Function<B, C>> f) {
    return a.flatMap(ax -> b.map(bx -> f.apply(ax).apply(bx)));
}
```

它与你用于 lift2 的样式相同。因此 map2 方法将如下所示：

```
public static <A, B, C> Result<C> map2(Result<A> a,
                                       Result<B> b,
                                       Function<A, Function<B, C>> f) {
    return lift2(f).apply(a).apply(b);
}
```

这种函数的一个常见用法是调用方法或构造函数并以其他函数或方法返回的 Result 类型为参数。以前面的 ToonMail 为例。要为 Toon map 填充数据，你可以通过让用户使用以下方法在控制台上输入名、姓和邮箱地址来构造 toon：

```
static Result<String> getFirstName() {
    return success("Mickey");
}

static Result<String> getLastName() {
    return success("Mickey");
}

static Result<String> getMail() {
    return success("mickey@disney.com");
}
```

真正的实现将是不同的，但你仍然需要学会如何在控制台以函数式的方式获取

输入。现在，你将使用这些模拟实现。

你可以创建一个如下所示的 Toon 来使用这些实现：

```
Function<String, Function<String, Function<String, Toon>>> createPerson =
    x -> y -> z -> new Toon(x, y, z);
Result<Toon> toon2 = lift3(createPerson)
    .apply(getFirstName())
    .apply(getLastName())
    .apply(getMail());
```

但是你达到了抽象的极限。你可能需要调用多于三个参数的方法或构造函数。在这种情况下，你可以使用以下模式：

```
Result<Toon> toon = getFirstName()
    .flatMap(firstName -> getLastName()
        .flatMap(lastName -> getMail()
            .map(mail -> new Toon(firstName, lastName, mail))));
```

这种模式有两个优点：

- 可以使用任意数量的参数。
- 不必定义一个函数。

请注意，你可以使用 lift3 而无须分别定义函数，但是由于 Java 的类型推断能力不强，你必须指定类型：

```
Result<Toon> toon2 =
    lift3((String x) -> (String y) -> (String z) -> new Toon(x, y, z))
        .apply(getFirstName())
        .apply(getLastName())
        .apply(getMail());
```

你的新模式有时被称为解析式 (comprehension)。一些语言为这样的结构提供了语法糖，大致相当于：

```
for {
    firstName in getFirstName(),
    lastName in getLastName(),
    mail in getMain()
} return new Toon(firstName, lastName, mail)
```

Java 中没有这种语法糖，但即使没有也很容易就能做到。不过要注意调用 flatMap 或 map 是嵌套的。首先调用第一个方法（或从一个 Result 实例开始），对每个新调用使用 flatMap，最后把调用映射到你打算使用的构造函数或方法。例

如有 5 个 Result 实例时，使用以下方式来调用接收 5 个参数的方法：

```
Result<Integer> result1 = success(1);
Result<Integer> result2 = success(2);
Result<Integer> result3 = success(3);
Result<Integer> result4 = success(4);
Result<Integer> result5 = success(5);
Result<Integer> result = result1
    .flatMap(p1 -> result2
        .flatMap(p2 -> result3
            .flatMap(p3 -> result4
                .flatMap(p4 -> result5
                    .map(p5 -> compute(p1, p2, p3, p4, p5))))));

private int compute(int p1, int p2, int p3, int p4, int p5) {
    return p1 + p2 + p3 + p4 + p5;
}
```

这个例子有些生硬，但它展示了如何能够扩展模式。然而，最后的调用（最深的嵌套）实际上是 map 而非 flatMap，这一点并不是模式所固有的。这是因为最后一个方法（compute）返回了一个原始值。如果它返回一个 Result，你就需要使用 flatMap 而不是 map。但是由于最后一个方法通常是一个构造函数，并且构造函数总是返回原始值，因而你会经常发现自己在最后一个方法的调用上使用 map。

7.6 总结

- 有必要表示由于错误而导致的数据缺失。Option 类型并不支持。
- Either 类型允许你表示一种类型（Right）或另一种（Left）类型的数据。
- Either 可以像 Option 那样 map 或 flatMap，但它可以有两边（right 或 left）。
- Either 可以通过使一侧（Left）始终表示相同的类型（RuntimeException）而有所侧重。我们称这个有所侧重的 Either 类型为 Result。成功由 Success 子类型表示，失败由 Failure 子类型表示。
- 一种使用 Result 类型的方式是在包装的值存在时获取它，否则使用提供的默认值。
- 默认类型如果不是字面量，必须进行延迟计算。
- 复合 Option（表示可选数据）与 Result（表示数据或错误）很没意思。通过往 Result 中添加一个 Empty 子类型而使 Option 类型变得毫无价值，

可以降低这个用例的难度。

- 如果需要，可以映射 `Failure`，例如使错误消息更加明确。
- 几个静态工厂方法简化了在各种情况下创建 `Result`，例如使用可空数据或条件数据，由数据和必须满足的条件表示。
- 可以通过 `forEach` 方法将作用应用于 `Result`（虽然是以非函数式的方式）。
- 如果数据存在则应用作用，否则抛出异常。`forEachOrThrow` 方法用于处理此类情况。
- `forEach` 和 `forEachOrThrow` 方法是更通用的 `forEachOrException` 的特定情况。此方法应用作用（如果值存在）并返回 `Empty`（如果可以应用作用），或 `Success<RuntimeException>`（如果数据缺失）。
- 可以将从 A 到 B 的函数提升为（使用 `lift` 方法）从 `Result<A>` 到 `Result`。你可以将从 A 到 B 到 C 的函数提升为（通过 `lift2` 方法）从 `Result<A>` 到 `Result` 到 `Result<C>` 的函数。
- 可以使用解析式模式来复合任意数量的 `Result`。

8 列表处理进阶

本章要点

- 使用记忆化加速列表处理
- 复合 List 与 Result
- 在列表上实现按索引访问
- 展开列表
- 列表自动并行处理

在第 5 章中，你创建了作为第一个数据结构的单链表。你在那时还没掌握使其成为一个完整的数据处理工具所需的全部技术。你缺少的一个特别有用的工具，是一些表示生成可选数据的操作，或可以生成错误的操作。在第 6 章和第 7 章中，你学习了如何表示可选数据和错误。在本章中，你将学习如何复合用列表生成可选数据或错误的操作。

你还开发了一些与最优解还相差甚远的函数，如 `length`，我还说过你最终会学到更高效的技术。在本章中，你将学习如何实现这些技术。你还将学习如何自动化并行一些列表操作，以受益于当今计算机的多核架构。

8.1 length的问题

折叠一个列表涉及从一个值开始，连续地复合该值与列表中的每个元素。所需时间显然与列表的长度成正比。有什么办法能让这个操作更快吗？或者，至少，有没有办法让它显得更快？

作为折叠程序的示例，你在练习 5.9 的 `List` 中用以下实现创建了一个 `length` 方法：

```
public int length() {  
    return foldRight(this, 0, x -> y -> y + 1);  
}
```

在这个实现中，用对结果加 1 的操作来折叠列表。初始值为 0，而列表中每个元素的值都被忽略了。这种做法允许你对所有列表使用相同的定义。因为列表元素被忽略，所以结果与列表元素的类型无关。

你可以比较上述操作与计算整型列表总和的操作：

```
public static Integer sum(List<Integer> list) {  
    return list.foldRight(0, x -> y -> x + y);  
}
```

这里的主要区别在于 `sum` 方法只能使用整型，而 `length` 方法适用于任何类型。请注意，`foldRight` 只是抽象递归的一种方式。对于空列表来说，列表的长度可以定义为 0；而对于非空列表来说，长度可以定义为 1 加 `tail` 的长度。同样，整型列表的和可以递归地定义为空列表的 0，以及非空列表的 `head` 加上 `tail` 的总和。

还有可以同样应用于列表的其他操作，有些与列表元素的类型无关：

- 列表的哈希码可以简单地通过对其元素的哈希码求和来计算。因为哈希码是整型的（至少对于 Java 对象来说），所以此操作不依赖于对象的类型。
- 由 `toString` 方法返回的列表的字符串形式可以通过复合列表元素的 `toString` 表达式来得出。还是与元素的实际类型无关。

一些操作可能取决于元素类型的某些特性，而并不指定类型本身。例如，返回列表最大元素的 `max` 方法只要求类型为 `Comparable` 或 `Comparator` 即可。

8.1.1 性能问题

所有这些方法都可以用折叠来实现，但是这样的实现都有一个重要缺点：计算

结果所需的时间与列表的长度成正比。假设你有一个包含大约 100 万个元素的列表，并想查看长度。对元素计数似乎是唯一的方式（这正是折叠 `length` 方法所做的）。但是，如果你在列表中添加元素直到它达到了 100 万个，那么你肯定不会在每添加一个元素之后就对元素重新计数。

在这种情况下，你可以将元素的数量保存在某个地方，并在每次添加元素到列表时对其加 1。也许一开始你就不得不对一个非空列表计算一次，但仅此而已。这种技术是你在第 4 章中学到的：记忆化。问题是，哪里可以存储记忆值？答案很明显：列表本身。

8.1.2 记忆化的优点

维护列表元素的计数器需要一些时间，因此添加元素到列表中会比不保留计数稍慢。它似乎是以时间换时间。如果你创建了 100 万个元素的列表，那么你将失去 100 万次递增计数器所需的时间。然而作为补偿，获取列表长度所需的时间将接近于 0（并且明显为常量）。也许在增加计数器时失去的总时间将等于调用 `length` 的收益。但只要你不止一次调用 `length`，成效绝对显著。

8.1.3 记忆化的缺点

记忆化可以把在 $O(n)$ 时间（时间与元素数量成正比）的函数转换为 $O(1)$ 时间（常量时间）。这是一个巨大的优点，尽管由于它让插入元素稍慢，会有一个时间成本，但是插入速度一般不是一个大问题。

一个更重要的问题是内存空间的增加。实现了直接修改的数据结构没有这个问题。在可变列表中，什么都无法阻止你将列表长度记忆为仅占用 32 位的可变整型。但要是使用不可变列表，你需要记忆每个元素的长度。很难知道空间增加的确切大小，但是如果单链表的大小是每节点约 40 字节（对于节点本身来说），再加上 `head` 和 `tail` 的两个 32 位引用（在 32 位 JVM 上），这将导致每个元素耗费大约 100 个字节。在这种情况下，添加长度会导致空间增长略高于 30%。如果记忆值是引用，结果将会相同，例如记录 `Comparable` 对象列表的 `max` 或 `min`。在 64 位 JVM 上，由于引用的大小做了一些优化，因此更难计算，但是你知道我的意思。

对象引用的大小 有关 Java 7 和 Java 8 中对象引用大小的更多信息，请参阅 Oracle 关于 Compressed Oops (<http://mng.bz/TjY9>) 和 JVM 性能增强的文档 (<http://mng.bz/8X0o>)。

你可以自行决定是否要在数据结构中使用记忆化。如果经常调用函数并且不会为返回结果创建新对象，那么这样做可能是一个正确的选择。例如，`length` 和 `hashCode` 函数返回整型，而 `max` 和 `min` 函数返回现有对象的引用，因此它们可能是不错的备选。另一方面，`toString` 函数创建了必须被记忆的新字符串，这样可能会浪费大量的内存空间。另一个要考虑的因素是函数的调用频率。`length` 函数可能会比 `hashCode` 使用得更频繁，因为用列表作为 `map` 的键并不常见。

练习 8.1

创建一个记忆化版的 `length` 方法。它在 `List` 类中的签名为

```
public abstract int lengthMemoized();
```

答案 8.1

在 `Nil` 类中的实现与非记忆化的 `length` 方法完全相同：

```
public int lengthMemoized() {  
    return 0;  
}
```

要实现 `Cons` 版，必须先将记忆化字段添加到类中，并在构造函数中将其初始化：

```
private final int length;  
private Cons(A head, List<A> tail) {  
    this.head = head;  
    this.tail = tail;  
    this.length = tail.length() + 1;  
}
```

之后你可以实现 `lengthMemoized` 方法轻松地返回长度：

```
public int lengthMemoized() {  
    return length;  
}
```

这个版本比原来的版本要快得多。一件值得注意的趣事是 `length` 与 `isEmpty` 方法之间的关系。你可能倾向于认为 `isEmpty` 等价于 `length == 0`，但是虽然从逻辑的角度上看为真，但在实现与性能上可能会有巨大差异。

请注意,在 `Comparable` 类型的列表中记忆最大值或最小值也可以同样完成(虽然用的是一个静态方法),但如果你会从列表中删除最大或最小值,那就没什么用了。最小或最大元素经常用于按优先级检索元素。在这种情况下,元素的 `compareTo` 方法将比较它们的优先级。记忆优先级会让你立即知道哪个元素的优先级最高,但是鉴于你会经常删除相应的元素,它不会有太大帮助。这种用例需要一个不同的数据结构,你将在第 11 章中学习如何创建。

8.1.4 实际性能

我曾说过,你可以自行决定是否应该记忆 `List` 类的一些函数。一些实验应该有助于你做出决定。当使用记忆化时,在创建包含 100 万个整型的列表之前和之后测量可用的内存大小,结果显示有极小幅的增加。虽然这种测量方法不是非常精确,但是在两种情况下(有或没有记忆化),可用内存的平均减少量约为 22 MB,在 20 MB 到 25 MB 之间变化。这表明理论上增加 4 MB ($1\,000\,000 \times 4$ 字节)并不像预期那么大。另一方面,性能有了巨幅提升。调用 10 次不使用记忆化的 `length` 可能超过 200 毫秒。要是使用记忆化,时间为 0 (时间短至低于毫秒的精度)。

请注意,虽然添加元素会增加成本(`tail` 的长度加 1 并存储结果),但是由于 `tail` 的长度已被记忆,所以删除元素的成本为零。

如果不要记忆化,还有一种方式可以优化 `length` 方法。你可以不使用折叠,而是用循环和局部可变变量的命令式风格。以下是从 `Scala List` 类中借用的 `length` 实现:

```
public int length() {
    List<A> these = this;
    int len = 0;
    while (!these.isEmpty()) {
        len += 1;
        these = these.tail();
    }
    return len;
}
```

虽然它的风格看起来并不很函数式,但是这种实现与函数式编程的定义完全兼容。这是一个在外界没有任何可观测效果的纯函数。主要的问题是,它的速度只比基于折叠的实现快了 5 倍,对于非常庞大的列表而言,记忆化实现的速度可以提高数百万倍。

8.2 复合List和Result

在上一章中，你看到了 `Result` 和 `List` 的数据结构非常相似，主要区别在于基数（cardinality），但它们共享了一些最重要的方法，例如 `map`、`flatMap`，甚至还有 `foldLeft` 和 `foldRight`。

你看到列表可以与列表复合，还有结果与结果复合。现在，你将看到如何复合列表与结果。

8.2.1 List 中返回 Result 的方法

在这一点上，你已经注意到了我试图避免直接访问结果和列表的元素。如果列表为 `Nil`，访问列表的 `head` 或 `tail` 会抛出异常，而这是函数式编程中可能发生的最糟糕的事情之一。但是，你也看到了在结果为 `failure` 或 `empty` 的情况下，可以通过提供默认值来安全地访问 `Result` 中的值。访问列表的 `head` 时，你可以照猫画虎吗？并不完全如此，但是你可以返回一个 `Result`。

练习 8.2

在 `List<A>` 中实现 `headOption` 方法以返回一个 `Result<A>`。

提示

在 `List` 中使用以下抽象方法声明，并在每个子类中实现：

```
public abstract Result<A> headOption();
```

请注意，该方法被称为 `headOption`，表示值是可选的，虽然你用的是该类型的 `Result`。

答案 8.2

`Nil` 类的实现返回 `Empty`：

```
public Result<A> headOption() {  
    return Result.empty();  
}
```

`Cons` 的实现返回一个保存着 `head` 值的 `Success`：

```
public Result<A> headOption() {  
    return Result.success(head);  
}
```

练习 8.3

创建一个 `lastOption` 方法以返回列表中最后一个元素的 `Result`。

提示

不要使用显式递归，而是尝试使用基于第5章中开发的方法。你应该能够在 `List` 类中定义一个方法。

答案 8.3

一个通用的方案是使用显式递归：

```
public Result<A> lastOption() {
    return isEmpty()
        ? Result.empty()
        : tail().isEmpty()
            ? Result.success(head())
            : tail().lastOption();
}
```

这个方案有几个问题。它是基于栈的递归，所以你应该使其基于堆，另外还得处理空列表的情况，否则 `tail().lastOption()` 会抛出一个 `NPE`。

不过你可以简单地使用折叠，它为你抽象了递归！需要做的全部就是创建正确的折叠函数。你需要在最后一个值存在时始终保留着它。以下是可供使用的函数：

```
Function<Result<A>, Function<A, Result<A>>>> f =
    x -> y -> Result.success(y);
```

或使用方法引用：

```
Function<Result<A>, Function<A, Result<A>>>> f =
    x -> Result::success;
```

接下来你只需使用 `Result.Empty` 作为单位元来 `foldLeft` 列表：

```
public Result<A> lastOption() {
    return foldLeft(Result.empty(), x -> Result::success);
}
```

练习 8.4

你可以在 `List` 类中用一个实现替换 `headOption` 方法吗？这个实现的优点和缺点都是什么？

答案 8.4

可以创建如下实现：

```
public Result<A> headOption() {  
    return foldRight(Result.empty(), x -> y -> Result.success(x));  
}
```

唯一的优点是，如果你喜欢这种方式，它会更有趣。在设计 `lastOption` 的实现时，你知道必须遍历列表才能找到最后一个元素。要找到第一个元素，你不需要遍历列表。在这里使用 `foldRight` 与反转列表完全相同，然后遍历结果以找到最后一个元素（这是原始列表的第一个元素）。效率并不高！顺便说一下，这正是 `lastOption` 方法找到最后一个元素的办法：反转列表并获取结果的第一个元素。所以除了有趣，真的没有理由来用这个来实现。

8.2.2 将 List<Result> 转换为 Result<List>

当列表包含一些计算的结果时，它一般是 `List<Result>`。例如，在 `T` 的列表上映射从 `T` 到 `Result<U>` 的函数，将生成 `Result<U>` 的列表。这些值通常必须与以 `List<T>` 为参数的函数复合。也就是说，你需要一种将 `List<Result<U>>` 转换为 `List<U>` 的方式，它与 `flatMap` 需要展平的情况相同，就是涉及的两种不同的数据类型差异很大：`List` 和 `Result`。你可以对这种转换应用几种对策：

- 丢弃所有 `failure` 或 `empty` 的结果，并从剩余的 `success` 列表中生成 `U` 列表。如果列表中没有 `success`，结果可能只是一个空的 `List`。
- 丢弃所有 `failure` 或 `empty` 的结果，并从剩余的 `success` 列表中生成 `U` 列表。如果列表中没有 `success`，结果就是 `Failure`。
- 确定所有元素都必须是 `success`，才能使整个操作成功。如果所有的结果都是 `success`，用它构造一个 `U` 列表，并将其作为 `Success<List<U>>` 返回，否则返回 `Failure<List<U>>`。

第一种方案对应于所有结果都是可选的结果列表。第二种方案意味着想要结果成功，在列表中应该至少有一个 `success`。第三种方案对应于所有结果都为必需的情况。

练习 8.5

编写一个名为 `flattenResult` 的方法，以 `List<Result<A>>` 为参数并返回一个 `List<A>`，其中包含原始列表中所有值为 `success` 的元素，并忽略 `failure` 和 `empty` 值。这是一个 `List` 中的静态方法，具有以下签名：

```
public static <A> List<A> flattenResult(List<Result<A>> list)
```

尽量不要使用显式递归，而是复合 `List` 和 `Result` 类的方法。

提示

为方法选择的名称要表明你要做什么。

答案 8.5

要解答这道练习题，你可以用 `foldRight` 方法来折叠列表，使用一个生成列表的列表的函数。每个 `Success` 都将被转换为包含该值的单个元素的列表，而每个 `Failure` 或 `Empty` 都将被转换为一个空列表。函数如下所示：

```
Function<Result<A>, Function<List<List<A>>, List<List<A>>>> f =  
    x -> y -> y.cons(x.map(List::list).getOrElse(list()));
```

一旦有了这个函数，你可以用它向右折叠列表，生成一个值的列表的列表，其中有一些元素是空列表：

```
list.foldRight(list(), f)
```

剩下要做的就是 `flatten` 这个结果。完整的方法如下：

```
public static <A> List<A> flattenResult(List<Result<A>> list) {  
    return flatten(list.foldRight(list(), x -> y ->  
        y.cons(x.map(List::list).getOrElse(list()))));  
}
```

请注意，这并不是最高效的方法。主要还是把它当作一个练习。

练习 8.6

编写将 `List<Result<T>>` 复合为 `Result<List<T>>` 的 `sequence` 函数。如果原始列表中的所有值均为 `Success` 实例，则为 `Success<List<T>>`，否则就是 `Failure<List<T>>`。它的签名如下：

```
public static <A> Result<List<A>> sequence(List<Result<A>> list)
```

提示

还是使用 `foldRight` 方法，而不是显式递归。还需要定义在 `Result` 类中的 `map2` 方法。

答案 8.6

这是使用 `foldRight` 和 `map2` 的实现：

```
public static <A> Result<List<A>> sequence(List<Result<A>> list) {
    return list.foldRight(Result.success(List.list()),
        x -> y -> Result.map2(x, y, a -> b -> b.cons(a)));
}
```

请注意，这个实现把一个空的 `Result` 作为 `Failure` 处理，并返回遇到的第一个 `failure`，它可能是 `Failure` 或 `Empty`。这可能就是你要的，也可能不是。如果你坚持认为 `Empty` 表示可选数据，那就需要先过滤列表以删除 `Empty` 元素：

```
public static <A> Result<List<A>> sequence2(List<Result<A>> list) {
    return list.filter(a -> a.isSuccess() || a.isFailure())
        .foldRight(Result.success(List.list()),
            x -> y -> Result.map2(x, y, a -> b -> b.cons(a)));
}
```

最后你应该把在 `List` 类中删除空元素抽象为一个单独的方法。但是对于本书的其余部分，我们将在 `sequence` 方法的上下文中继续把 `Empty` 作为 `Failure` 处理。

练习 8.7

定义一个更通用的 `traverse` 方法，遍历 `A` 列表，同时应用从 `A` 到 `Result` 的函数并生成 `Result<List>`。它的签名如下：

```
public static <A, B> Result<List<B>> traverse(List<A> list,
    Function<A, Result<B>> f)
```

然后根据 `traverse` 定义新版本的 `sequence`。

提示

不要使用递归。应该优先使用 `foldRight` 方法，它为你抽象了递归。

答案 8.7

首先定义 `traverse` 方法：

```
public static <A, B> Result<List<B>> traverse(List<A> list,
                                             Function<A, Result<B>> f) {
    return list.foldRight(Result.success(List.list()),
                          x -> y -> Result.map2(f.apply(x), y, a -> b -> b.cons(a)));
}
```

然后你可以通过 `traverse` 重新定义 `sequence` 方法：

```
public static <A> Result<List<A>> sequence(List<Result<A>> list) {
    return traverse(list, x -> x);
}
```

8.3 抽象常见列表用例

`List` 数据类型有许多值得抽象的常见用例，因此你不必一次又一次地重复相同的代码。你会经常发现自己找到了可以通过复合基本函数来实现的新用例。你永远也不要对把这些用例作为新函数并入 `List` 类中有丝毫犹豫。下面的练习展示了几种最常见的用例。

8.3.1 压缩和解压缩列表

压缩 (`zip`) 是通过合并相同索引的元素将两个列表合并为一个列表的过程。解压缩 (`unzip`) 是与此相反的过程，包括通过“解构” (deconstructing) 元素来生成两个列表，例如从一个方位列表中生成 `x` 和 `y` 两个坐标列表。

练习 8.8

编写一个 `zipWith` 方法，给定一个函数参数，复合两个不同类型的列表元素以产生一个新的列表。签名如下：

```
public static <A, B, C> List<C> zipWith(List<A> list1, List<B> list2,
                                         Function<A, Function<B, C>> f)
```

该方法接收一个 `List<A>` 和一个 `List`，并通过从 `A` 到 `B` 到 `C` 的函数生成 `List<C>`。

提示

压缩应限制于最短列表的长度。

答案 8.8

对这个练习来说，由于必须同时在两个列表上完成递归，所以你需要使用显式递归，没有任何可用的抽象能处理这个问题。解决方案如下：

```
public static <A, B, C> List<C> zipWith(List<A> list1, List<B> list2,
                                         Function<A, Function<B, C>> f) {
    return zipWith_(list(), list1, list2, f).eval().reverse();
}

private static <A, B, C> TailCall<List<C>> zipWith_(List<C> acc,
    List<A> list1, List<B> list2, Function<A, Function<B, C>> f) {
    return list1.isEmpty() || list2.isEmpty()
        ? ret(acc)
        : sus(() -> zipWith_(
            new Cons<>(f.apply(list1.head()).apply(list2.head()), acc),
            list1.tail(), list2.tail(), f));
}
```

使用空列表作为初始累加器来调用 `zipWith_` 辅助方法。如果两个参数列表之一为空，则停止递归并返回当前的累加器。否则，通过将函数应用于两个列表的 `head` 值来计算新值，并使用两个参数列表的 `tail` 递归调用辅助函数。

练习 8.9

上一个练习包含了用两个列表中索引匹配的元素创建一个列表。编写一个 `product` 方法，用于生成一个列表，它包含了从两个列表中获取的所有可能组合。换句话说，给定两个列表 `list("a", "b", "c")`、`list("d", "e", "f")` 和字符串连接，两个列表的积（`product`）应为 `List("ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf")`。

提示

此练习无须使用显式递归。

答案 8.9

该方案类似于第 7 章中用于复合 `Result` 的解析式模式。唯一的区别是它生成的组合与列表中元素数量的积一样多，而对于复合 `Result` 来说，数量总是限定为 1。

```
public static <A, B, C> List<C> product(List<A> list1, List<B> list2,
                                         Function<A, Function<B, C>> f) {
    return list1.flatMap(a -> list2.map(b -> f.apply(a).apply(b)));
}
```


请注意，这种方式可以合并两个以上的列表。唯一的问题是组合的数量会呈指数级增长。

`product` 和 `zipWith` 的常见用例之一是使用构造函数来复合函数。以下是使用元组构造函数的示例：

```
List.product(List.list(1, 2, 3), List.list(4, 5, 6),
              x -> y -> new Tuple<>(x, y));
List.zipWith(List.list(1, 2, 3), List.list(4, 5, 6),
             x -> y -> new Tuple<>(x, y));
```

第一行生成的列表中包含了由两个列表的元素构建的所有可能的元组：

```
[(1,4), (1,5), (1,6), (2,4), (2,5), (2,6), (3,4), (3,5), (3,6), NIL]
```

第二行仅会生成由相同索引的元素构建而成的元组列表：

```
[(1,4), (2,5), (3,6), NIL]
```

你当然也可以用任何类的任何构造函数。（Java 的对象其实就是指定了名称的元组。）

练习 8.10

编写一个 `unzip` 静态方法将一个元组列表转换为列表元组。以下是它的签名：

```
<A, B> Tuple<List<A>, List<B>> unzip(List<Tuple<A, B>> list)
```

提示

不要使用显式递归。简单地调用一下 `foldRight` 应该就可以了。

答案 8.10

你需要使用两个空列表组成的元组作为单位元来 `foldRight` 列表：

```
public static <A,B> Tuple<List<A>, List<B>> unzip(List<Tuple<A, B>> list) {
    return list.foldRight(new Tuple<>(list(), list()),
                          t -> tl -> new Tuple<>(tl._1.cons(t._1), tl._2.cons(t._2)));
}
```

练习 8.11

泛化 `unzip` 函数，使其可以将任何类型的列表转换为列表元组，给定一个以列表类型的对象为参数的函数，并生成一个元组。例如，给定一个 `Payment` 实例列表，

你应该能够生成一个列表元组：一个包含用于付款的信用卡，另一个包含支付金额。

在 `List` 中用以下签名的实例方法来实现：

```
<A1, A2> Tuple<List<A1>, List<A2>> unzip(Function<A, Tuple<A1, A2>> f)
```

提示

答案与练习 8.10 基本相同。

答案 8.11

一件重要的事情是：函数被用了两次。为了避免应用两次该函数，你需要使用一个多行 `lambda`：

```
public <A1, A2> Tuple<List<A1>, List<A2>> unzip(Function<A,
                                         Tuple<A1, A2>> f) {
    return this.foldRight(new Tuple<>(list(), list()), a -> tl -> {
        Tuple<A1, A2> t = f.apply(a);
        return new Tuple<>(tl._1.cons(t._1), tl._2.cons(t._2));
    });
}
```

8.3.2 通过索引访问元素

就通过索引访问元素而言，单链表并非最佳结构，但有时需要用索引来访问。与往常一样，你应该将这个过程抽象为 `List` 的方法。

练习 8.12

编写一个 `getAt` 方法，它以一个索引为参数，并返回相应的元素。在索引超出范围的情况下，该方法不应该抛出异常。

提示

这回从一个显式的递归版本开始。然后尝试回答以下问题：

- 可以使用折叠吗？右折叠还是左折叠？
- 显式递归版本为什么更好？
- 你能找到一个解决问题的办法吗？

答案 8.12

显式递归方案很简单：

```

public Result<A> getAt(int index) {
    return index < 0 || index >= length()
        ? Result.failure("Index out of bound")
        : getAt_(this, index).eval();
}

private static <A> TailCall<Result<A>> getAt_(List<A> list, int index) {
    return index == 0
        ? TailCall.ret(Result.success(list.head()))
        : TailCall.sus(() -> getAt_(list.tail(), index - 1));
}

```

首先，你可以检查索引，看看它是否为正数，并且小于列表长度。如果不是，返回一个 Failure 即可。否则，调用辅助方法递归处理列表。该方法检查索引是否为 0。如果是，返回列表的 head。否则，用递减的索引和列表的 tail 来递归调用自己。

这似乎就是最好的递归方案了。可以用折叠吗？当然可以，它应该是左折叠。但这个方案有些棘手：

```

public Result<A> getAt(int index) {
    Tuple<Result<A>, Integer> identity =
        new Tuple<>(Result.failure("Index out of bound"), index);

    Tuple<Result<A>, Integer> rt = index < 0 || index >= length()
        ? identity
        : foldLeft(identity, ta -> a -> ta._2 < 0
            ? ta
            : new Tuple<>(Result.success(a), ta._2 - 1));
    return rt._1;
}

```

首先你需要定义单位元的值。由于该值必须同时保存结果和索引，它将是一个保存 Failure 的 Tuple。接下来可以检查索引的有效性。如果发现索引无效，则让临时结果 (rt) 等于 identity。否则向左折叠，传入的函数在索引值小于 0 时返回已计算的结果 (ta)，否则返回一个新的 Success。

这个方案看起来可能更巧妙，但并非如此，原因有三：

- 它一点儿也不易读。这可能有点儿主观，因此以你自己的看法为准。
- 你必须使用一个中间结果 (rt)，因为 Java 不能推断出正确的类型。如果你不相信，请尝试在最后一行将 rt 替换为它的值。
- 效率较低，因为即使找到了要搜索的值，它仍将继续折叠整个列表。

练习 8.13 (难, 可选)

找到一个方案, 使得基于折叠的版本一旦找到结果就会终止。

提示

为此你将需要一个特定版的 `foldLeft`, 还要一个特定版的 `Tuple`。

答案 8.13

首先, 你需要一个特定版的 `foldLeft`, 可以在找到折叠操作的吸收元 (或 “零” 元素) 时退出折叠。想象一个整型列表, 你想用乘法来折叠它。乘法的吸收元为 0。以下是 `List` 类中的 `foldLeft` 的短路 (或脱离) 版定义:

```
public abstract <B> B foldLeft(B identity, B zero,
                               Function<B, Function<A, B>> f);
```

零元 与此相似, 任意操作的吸收元有时被称为 “零”, 但请记住, 它并不总是等于 0。0 仅是乘法的吸收元而已。对于正整数求和而言, 吸收元是无穷大。

这是 `Cons` 的实现:

```
@Override
public <B> B foldLeft(B identity, B zero, Function<B, Function<A, B>> f) {
    return foldLeft(identity, zero, this, f).eval();
}

private <B> TailCall<B> foldLeft(B acc, B zero, List<A> list,
                                   Function<B, Function<A, B>> f) {
    return list.isEmpty() || acc.equals(zero)
        ? ret(acc)
        : sus(() -> foldLeft(f.apply(acc).apply(list.head()),
                               zero, list.tail(), f));
}
```

如你所见, 唯一的区别是如果发现累加器的值为 “零”, 则停止递归并且将其返回。

现在你需要为折叠找到一个零值。零值是一个 `Integer` 值等于 -1 (第一个值小于 0) 的 `Tuple<Result<A, Integer>>`。可以用标准的 `Tuple` 吗? 不行, 因为它必须有一个特殊的 `equals` 方法, 无论 `Result<A>` 是什么, 只要 `Integer` 值相等就返回 `true`。完整的方法如下:

```

public Result<A> getAt(int index) {
    class Tuple<T, U> {
        public final T _1;
        public final U _2;

        public Tuple(T t, U u) {
            this._1 = Objects.requireNonNull(t);
            this._2 = Objects.requireNonNull(u);
        }

        @Override
        public boolean equals(Object o) {
            if (!(o.getClass() == this.getClass()))
                return false;
            else {
                @SuppressWarnings("rawtypes")
                Tuple that = (Tuple) o;
                return _2.equals(that._2);
            }
        }
    }

    Tuple<Result<A>, Integer> zero =
        new Tuple<>>(Result.failure("Index out of bound"), -1);
    Tuple<Result<A>, Integer> identity =
        new Tuple<>>(Result.failure("Index out of bound"), index);
    Tuple<Result<A>, Integer> rt = index < 0 || index >= length()
        ? identity
        : foldLeft(identity, zero, ta -> a -> ta._2 < 0
            ? ta
            : new Tuple<>>(Result.success(a), ta._2 - 1));
    return rt._1;
}

```

请注意，为了使代码更简短，我省略了 hashCode 和 toString 方法。

现在一旦找到了要搜索的元素，折叠将自动停止。当然，你可以使用新的 foldLeft 方法来退出任何具有零元的计算。（请记住：零，不是 0。）

8.3.3 拆分列表

有时你需要在特定位置将列表拆分为两部分。虽然对于这种操作来说，单链表远非理想结构，但实现起来相对简单。拆分列表有几个用处，其中之一是使用多个线程并行处理各部分。

练习 8.14

编写一个 `splitAt` 方法，它以 `int` 为参数，在给定位置拆分列表并返回两个列表。不应该有任何的 `IndexOutOfBoundsException`。相反，小于 0 的索引应被视为 0，而高于 `max` 的索引应被视为索引的最大值。

提示

让方法显式递归。

答案 8.14

显式递归的方案很容易设计：

```
public Tuple<List<A>, List<A>> splitAt(int index) {
    return index < 0
        ? splitAt(0)
        : index > length()
            ? splitAt(length())
            : splitAt(list(), this.reverse(), this.length() - index).eval();
}

private TailCall<Tuple<List<A>, List<A>>> splitAt(List<A> acc,
                                                    List<A> list, int i) {
    return i == 0 || list.isEmpty()
        ? ret(new Tuple<>(list.reverse(), acc))
        : sus(() -> splitAt(acc.cons(list.head()), list.tail(), i - 1));
}
```

请注意，第一个方法使用递归来调整索引的值。但是不必使用 `TailCall`，因为它最多只会递归一次。第二个方法与 `getAt` 方法非常相似，区别在于首先会反转列表。该方法累加元素直到达到索引的位置，因此累加列表的顺序正确，但是列表的剩余部分还需要反转回来。

练习 8.15（如果你完成了练习 8.13，就没那么难了）

你能否想出一个使用折叠而不是显式递归的实现？

提示

遍历整个列表的实现很容易。遍历列表直到索引为止的实现要难得多，并且需要一个新的特殊脱离版 `foldLeft`，同时返回退出值和列表的剩余部分。

答案 8.15

遍历整个列表的方案如下：

```
public Tuple<List<A>, List<A>> splitAt(int index) {
    int ii = index<0?0: index >= length() ? length() : index;
    Tuple3<List<A>, List<A>, Integer> identity =
        new Tuple3<>(List.list(), List.list(), ii);
    Tuple3<List<A>, List<A>, Integer> rt =
        foldLeft(identity, ta -> a -> ta._3 == 0
            ? new Tuple3<>(ta._1, ta._2.cons(a), ta._3)
            : new Tuple3<>(ta._1.cons(a), ta._2, ta._3 - 1));
    return new Tuple<>(rt._1.reverse(), rt._2.reverse());
}
```

折叠的结果累加到第一个列表累加器中，直到到达索引为止（在调整索引值以避免索引超出范围之后）。一旦找到索引，列表继续遍历，但把剩余的值累加到第二个列表累加器中。

这个实现的问题在于通过累加剩余的值到第二个列表累加器，其实可以反转列表的这一部分。不仅没必要遍历列表的剩余部分，而且它还做了两次：一次以相反的顺序累加，另一次最终反转结果。为了避免出现这种情况，应该修改 `foldLeft` 的特殊“脱离”版，因此它不仅将返回退出的结果（吸收元或零元），还将返回列表的剩余部分。要实现这一点，需要更改签名以返回一个 `Tuple`：

```
public abstract <B> Tuple<B, List<A>> foldLeft(B identity, B zero,
    Function<B, Function<A, B>> f);
```

然后你需要修改 `Nil` 类的实现：

```
@Override
public <B> Tuple<B, List<A>> foldLeft(B identity, B zero,
    Function<B, Function<A, B>> f) {
    return new Tuple<>(identity, list());
}
```

最后，你必须修改 `Cons` 的实现以返回列表的剩余部分：

```
@Override
public <B> Tuple<B, List<A>> foldLeft(B identity, B zero,
    Function<B, Function<A, B>> f) {
    return foldLeft(identity, zero, this, f).eval();
}

private <B> TailCall<Tuple<B, List<A>>> foldLeft(B acc, B zero,
    List<A> list, Function<B, Function<A, B>> f) {
```

```

return list.isEmpty() || acc.equals(zero)
  ? ret(new Tuple<>(acc, list))
  : sus(() -> foldLeft(f.apply(acc).apply(list.head()),
                      zero, list.tail(), f));
}

```

现在你可以用这个特殊的 foldLeft 方法重写 splitAt 方法：

```

public Tuple<List<A>, List<A>> splitAt(int index) {

    class Tuple3<T, U, V> {

        public final T _1;
        public final U _2;
        public final V _3;

        public Tuple3(T t, U u, V v) {
            this._1 = Objects.requireNonNull(t);
            this._2 = Objects.requireNonNull(u);
            this._3 = Objects.requireNonNull(v);
        }

        @Override
        public boolean equals(Object o) {
            if (!o.getClass() == this.getClass())
                return false;
            else {
                @SuppressWarnings("rawtypes")
                Tuple3 that = (Tuple3) o;
                return _3.equals(that._3);
            }
        }
    }

    Tuple3<List<A>, List<A>, Integer> zero =
        new Tuple3<>(list(), list(), 0);
    Tuple3<List<A>, List<A>, Integer> identity =
        new Tuple3<>(list(), list(), index);
    Tuple<Tuple3<List<A>, List<A>, Integer>, List<A>> rt = index <= 0
        ? new Tuple<>(identity, this)
        : foldLeft(identity, zero, ta -> a -> ta._3 < 0
            ? ta
            : new Tuple3<>(ta._1.cons(a), ta._2, ta._3 - 1));
    return new Tuple<>(rt._1._1.reverse(), rt._2);
}

```

你在此又需要一个特殊的 Tuple3 类，它拥有特殊的 equals 方法以在第三个元素相等时返回 true，而不考虑前两个元素。请注意，不必反转第二个结果列表。

什么时候不用折叠

折叠能用并不意味着你就应该这样做。前面的练习仅仅是：练习。作为函数式库的设计师，你需要选择最高效的实现。

函数式库必须具有函数式接口，并且必须遵守函数式编程的要求，这意味着所有函数都必须是没有副作用的纯函数，并且都必须遵守引用透明。库中发生的事情便与外界无关了。一个像 Java 那样面向命令式语言的函数库可以与面向函数的语言的编译器相提并论。编译后的代码永远是命令式的，因为只有这样计算机才能理解。函数式库提供了更多的选择。一些函数可以用函数式的风格实现，其他函数可以用命令式的风格实现，没有关系。拆分单链表或通过索引查找其元素用命令式的实现比函数式更快更容易，因为单链表并不适合这种操作。

最函数式的方法可能不是基于折叠来实现这些函数，而是避免实现它们。如果需要这些函数的函数式实现的结构，那么最好的做法就是创建特定的结构，你将在第 10 章中看到。

8.3.4 搜索子列表

列表的一个常见用例是，搜索列表是否包含在另一个（较长的）列表中。换句话说，你想知道一个列表是否是另一个列表的子列表。

练习 8.16

实现一个 `hasSubList` 方法来检查一个列表是否是另一个的子列表。例如，列表 `(3, 4, 5)` 是 `(1, 2, 3, 4, 5)` 的子列表，但不是 `(1, 2, 4, 5, 6)` 的子列表。将其实现为具有如下签名的静态方法：

```
public static <A> boolean hasSubsequence(List<A> list, List<A> sub)
```

提示

你首先需要实现一个 `startsWith` 方法来确定列表是否以子列表开头。一旦完成，你就可以从列表的每个元素开始递归检查该方法。

答案 8.16

一个显式递归的 `startsWith` 方法可以如下实现：

```
public static <A> Boolean startsWith(List<A> list, List<A> sub) {
    return sub.isEmpty()
        ? true
        : list.isEmpty()
            ? false
            : list.head().equals(sub.head())
                ? startsWith(list.tail(), sub.tail())
                : false;
}
```

这是一个基于栈的版本，可以使用 `TailCall` 转换为基于堆的版本：

```
public static <A> Boolean startsWith(List<A> list, List<A> sub) {
    return startsWith_(list, sub).eval();
}
```

```
public static <A> TailCall<Boolean> startsWith_(List<A> list,
                                                List<A> sub) {
    return sub.isEmpty()
        ? ret(Boolean.TRUE)
        : list.isEmpty()
            ? ret(Boolean.FALSE)
            : list.head().equals(sub.head())
                ? sus(() -> startsWith_(list.tail(), sub.tail()))
                : ret(Boolean.FALSE);
}
```

在那之后实现 `hasSubList` 就很直观了：

```
public static <A> boolean hasSubList(List<A> list, List<A> sub) {
    return hasSubList_(list, sub).eval();
}

public static <A> TailCall<Boolean> hasSubList_(List<A> list, List<A> sub) {
    return list.isEmpty()
        ? ret(sub.isEmpty())
        : startsWith(list, sub)
            ? ret(true)
            : sus(() -> hasSubList_(list.tail(), sub));
}
```

8.3.5 使用列表的其他函数

可以开发许多其他有用的函数来使用列表。以下练习将为你在这个领域提供一些实践。请注意，推荐的答案当然不是唯一的。请随意开发自己的解决方案。

练习 8.17

创建一个以从 A 到 B 的函数为参数并返回 Map 的 `groupBy` 方法，其键为把函数应用于列表中每个元素的结果，而值为与每个键相对应的元素列表。换句话说，给定如下 `Payment` 列表，

```
public class Payment {
    public final String name;
    public final int amount;

    public Payment(String name, int amount) {
        this.name = name;
        this.amount = amount;
    }
}
```

以下代码应该创建一个包含键 - 值对的 Map，其中的每个键都是一个名称，而与之相对应的值是相应人员的 `Payment` 列表：

```
Map<String, List<Payment>> map = list.groupBy(x -> x.name);
```

提示

使用上一章介绍的函数式 Map 包装器。这次尝试先创建一个命令式的版本，然后根据折叠创建一个函数式的版本。你更喜欢哪一个？

答案 8.17

这是一个命令式的版本。没有什么可说的，因为它只是传统的命令式代码并有一个局部可变状态：

```
public <B> Map<B, List<A>> groupByImperative(Function<A, B> f) {
    List<A> workList = this;
    Map<B, List<A>> m = Map.empty();
    while (!workList.isEmpty()) {
        final B k = f.apply(workList.head());
        List<A> rt = m.get(k).getOrElse(list()).cons(workList.head());
        m = m.put(k, rt);
        workList = workList.tail();
    }
    return m;
}
```

请注意，这个实现是完美的函数式的，因为在方法外部看不见状态改变。但是代码风格是极其命令式的，有一个 `while` 循环和局部变量。

这是一个更加函数式的版本，使用了折叠：

```
public <B> Map<B, List<A>> groupBy(Function<A, B> f) {
    return foldRight(Map.empty(), t -> mt -> {
        final B k = f.apply(t);
        return mt.put(k, mt.get(k).getOrElse(list()).cons(t));
    });
}
```

你可以选择自己喜欢的风格。第二个版本显然更紧凑。但主要的优点是更好地表达了意图。groupBy 是一个折叠。选择命令式风格是对折叠的重新实现，而选择函数式风格则是重用抽象。

练习 8.18

编写一个 unfold 方法，以一个起始元素 S 和一个从 S 到 Result<Tuple<A, S>> 的函数 f 为参数，并通过依次将 f 应用于 S 值，在结果为 Success 时生成 List<A>。换句话说，以下代码应该生成从 0 到 9 的整数列表：

```
List.unfold(0, i -> i < 10
    ? Result.success(new Tuple<>(i, i + 1))
    : Result.empty());
```

答案 8.18

一个简单的非栈安全的递归版本很容易实现：

```
public static <A, S> List<A> unfold_(S z,
    Function<S, Result<Tuple<A, S>>> f) {
    return f.apply(z).map(x ->
        unfold_(x._2, f).cons(x._1)).getOrElse(list());
}
```

不幸的是，虽然这个方案很巧妙，但是它会有多于 1000 次的递归步骤并把栈击穿。为了解决这个问题，你可以创建一个尾递归版本，并用 TailCall 类在堆上递归：

```
public static <A, S> List<A> unfold(S z,
    Function<S, Result<Tuple<A, S>>> f) {
    return unfold(list(), z, f).eval().reverse();
}

private static <A, S> TailCall<List<A>> unfold(List<A> acc, S z,
    Function<S, Result<Tuple<A, S>>> f) {
    Result<Tuple<A, S>> r = f.apply(z);
```

```
Result<TailCall<List<A>>> result =
    r.map(rt -> sus(() -> unfold(acc.cons(rt._1), rt._2, f)));
return result.getOrElse(ret(acc));
}
```

但是请注意，这样做反转了列表。对于小列表而言，这可能不是一个大问题，但是对于大列表而言，这就是一个大问题。在这种情况下，恢复到命令式风格可能是一个选择。

练习 8.19

编写一个以两个整型为参数的 `range` 方法，并生成大于等于第一个且小于第二个整型的所有整型的列表。

提示

当然，你应该用已定义过的方法。

答案 8.19

如果你重用练习 8.18 中的方法，这很简单：

```
public static List<Integer> range(int start, int end) {
    return List.unfold(start, i -> i < end
        ? Result.success(new Tuple<>(i, i + 1))
        : Result.empty());
}
```

练习 8.20

创建一个 `exists` 方法，接收一个从 `A` 到 `Boolean` 的函数以表示条件，如果列表至少包含一个满足该条件的元素，则返回 `true`。不要使用显式递归，而是尝试使用已经定义过的方法。

提示

不必为列表中的所有元素计算条件。一旦找到满足条件的第一个元素，该方法就应该立即返回。

答案 8.20

递归方案可以如下定义：

```
public boolean exists(Function<A, Boolean> p) {
```

```
return p.apply(head()) || tail().exists(p);
}
```

由于 `||` 操作符惰性地计算其第二个参数，一旦找到满足断言 `p` 所表达的条件元素，递归过程就会停止。但这是一种非尾递归的基于栈的方法，如果列表很长，并且在前 1000 个或 2000 个元素中没有找到满足的元素，那么它就会把栈击穿。顺便提一句，它也会在列表为空时抛出一个异常，所以你需要在 `List` 类中定义一个抽象方法，并在 `Nil` 子类中写一个特殊的实现。

一个更好的方案是通过零元参数重用 `foldLeft` 方法：

```
public boolean exists(Function<A, Boolean> p) {
    return foldLeft(false, true, x -> y -> x || p.apply(y))._1;
}
```

练习 8.21

创建一个 `forAll` 方法，它接收一个从 `A` 到 `Boolean` 的函数用以表示条件，如果列表中的所有元素都满足此条件，则返回 `true`。

提示

不要使用显式递归。再说一次，你并不总是需要对列表中的所有元素计算条件。`forAll` 方法将非常类似于 `exists` 方法。

答案 8.21

这个方案与 `exists` 方法非常接近，只有两个不同之处：反转了单位元和零元的值，`Boolean` 运算符是 `&&` 而不是 `||`：

```
public boolean forAll(Function<A, Boolean> p) {
    return foldLeft(true, false, x -> y -> x && p.apply(y))._1;
}
```

请注意重用 `exists` 方法也是另一种办法：

```
public boolean forAll(Function<A, Boolean> p) {
    return !exists(x -> !p.apply(x));
}
```

这个方法检查是否存在不符合相反条件的元素。

8.4 自动并行处理列表

大多数应用于列表的计算都会使用折叠。折叠涉及了应用与列表中的元素一样多的操作。对于很长的列表和持久的操作，折叠可能会需要相当长的时间。由于当今的大多数计算机都配备了多核处理器，你可能会试图找到一种使计算机并行处理列表的办法。

为了将折叠并行化，你只需要做一件事情（当然也要多核处理器）：允许你重新复合每个并行计算结果的一个额外操作。

8.4.1 并非所有的计算都可以并行化

以整型列表为例。找到所有整数的平均值并不能直接并行化。你可以将列表分为4个部分（如果你有一台4个处理器的计算机），并计算每个子列表的平均值。但是，你无法从子列表的平均值中计算出整个列表的平均值。

另一方面，计算列表的平均值意味着计算所有元素的总和，然后除以元素的数量。计算总和可以很容易地并行化：计算子列表的和，之后计算结果的总和。

这是一个非常特别的例子，用于折叠（求和）的操作与用于合并子列表结果的操作相同。而实际上并非总是如此。举一个把字符列表用加到字符串的方式折叠的例子。要合并中间的结果，你需要一个不同的操作：字符串连接。

8.4.2 将列表拆分为子列表

首先，你需要将列表拆分为子列表，这个操作需要自动执行。一个重要的问题是你应该得到多少个子列表。乍一看，你可能会认为每个可用的处理器对应一个子列表会比较理想，但并非完全如此。处理器数量（或更确切地说，逻辑内核数量）并不是最重要的因素。还有一个更重要的问题：所有子列表的计算会花费相同的时间吗？很可能不会，但这取决于计算的类型。如果由于你决定将4个线程专门用于并行处理，而要将列表拆分为4个子列表，某些线程可能会很快完成，而其他线程可能会需要进行更长时间的计算。这将会破坏并行化的好处，因为它可能导致大部分的计算任务都由某一个线程处理。

一个更好的办法是将列表拆分为大量子列表，然后把每个子列表都提交到一个线程池中。这样的话，一旦线程完成了一个子列表的处理，就会着手处理一个新的子列表。所以第一个任务是创建一个将列表拆分为子列表的方法。

练习 8.22

编写一个 `divide(int depth)` 方法，将一个列表拆分为多个子列表。该列表将被拆分为两部分，每个子列表递归拆分为两部分，`depth` 参数表示递归的步骤数。这个方法将在 `List` 父类中实现，签名如下：

```
List<List<A>> divide(int depth)
```

提示

首先你要定义一个新版的 `splitAt` 方法，该方法返回列表的列表而非 `Tuple<List, List>`。我们称这个方法为 `splitListAt`，并给定以下签名：

```
List<List<A>> splitListAt(int i)
```

答案 8.22

`splitListAt` 方法是一个显式的递归方法，通过使用 `TailCall` 类来实现栈安全：

```
public List<List<A>> splitListAt(int i) {
    return splitListAt(list(), this.reverse(), i).eval();
}

private TailCall<List<List<A>>> splitListAt(List<A> acc,
                                             List<A> list, int i) {
    return i == 0 || list.isEmpty()
        ? ret(List.list(list.reverse(), acc))
        : sus(() -> splitListAt(acc.cons(list.head()), list.tail(), i - 1));
}
```

当然这个方法总是会返回包含两个列表的列表。接下来可以如下定义 `divide` 方法：

```
public List<List<A>> divide(int depth) {
    return this.isEmpty()
        ? list(this)
        : divide(list(this), depth);
}

private List<List<A>> divide(List<List<A>> list, int depth) {
    return list.head().length() < depth || depth < 2
        ? list
        : divide(list.flatMap(x -> x.splitListAt(x.length() / 2)), depth / 2);
}
```


请注意，不必让这个方法栈安全，因为递归步骤的数量将只是 $\log(\text{length})$ 。换句话说，你永远不会有足够的堆内存来保存一个足够长的列表以发生栈溢出。

8.4.3 并行处理子列表

为了并行处理子列表，你会需要一个待执行方法的特殊版本，它将接收一个额外的参数 `ExecutorService`，其中配置了要并行使用的线程数。

练习 8.23

在 `List<A>` 中创建一个 `parFoldLeft` 方法，它将接收与 `foldLeft` 相同的参数并加上 `ExecutorService`，以及一个从 `B` 到 `B` 到 `B` 并返回 `Result<List>` 的函数。额外的函数将用于复合子列表的结果。方法的签名如下：

```
public<B> Result<B> parFoldLeft(ExecutorService es, B identity,
                                Function<B, Function<A, B>> f, Function<B, Function<B, B>> m)
```

答案 8.23

首先，你需要定义要使用的子列表数量，并相应地拆分列表：

```
final int chunks = 1024;
final List<List<A>> dList = divide(chunks);
```

接下来，你将用一个函数来映射子列表的列表，该函数会向 `ExecutorService` 提交一个任务。这个任务包含了折叠每个子列表并返回一个 `Future` 实例。`Future` 实例列表映射到一个函数上，该函数会调用每个 `Future` 的 `get` 方法，以生成一个结果列表（每个子列表都有一个）。请注意，你必须捕获潜在的异常。

最后，通过第二个函数折叠结果列表，并且把结果返回到一个 `Result.Success` 中。在出现异常的情况下，返回一个 `Failure`。

```
try {
    List<B> result = dList.map(x -> es.submit(() -> x.foldLeft(identity,
                                                                    f))).map(x -> {
        try {
            return x.get();
        } catch (InterruptedException | ExecutionException e) {
            throw new RuntimeException(e);
        }
    });
    return Result.success(result.foldLeft(identity, m));
} catch (Exception e) {
```

```
return Result.failure(e);
}
```

你可以在随书附带的代码 (<https://github.com/fpinjava/fpinjava>) 中找到该方法的基准测试程序示例。该基准包括以非常慢的算法计算 35 000 个 1 到 30 之间的随机数的斐波那契值 10 次。在 4 核的苹果电脑上，并行的版本在 22 秒内执行完毕，而串行的版本需要 83 秒。

练习 8.24

虽然可以通过折叠来实现映射（并且可以因此受益于自动并行化），但是它也可以不用折叠就实现并行。这也许是能在列表中实现的最简单的自动化并行处理。创建一个 `parMap` 方法，它会自动将给定的函数并行应用于列表中的所有元素。方法的签名如下：

```
public <B> Result<List<B>> parMap(ExecutorService es, Function<A, B> g)
```

提示

其实这个练习几乎没有什么可做的。只需将每个函数的应用提交给 `ExecutorService`，并从每个相应的 `Callable` 中获取结果即可。

答案 8.24

答案如下：

```
public <B> Result<List<B>> parMap(ExecutorService es, Function<A, B> g) {
    try {
        return Result.success(this.map(x -> es.submit(() -> g.apply(x)))
                                .map(x -> {
                                    try {
                                        return x.get();
                                    } catch (InterruptedException | ExecutionException e) {
                                        throw new RuntimeException(e);
                                    }
                                }
                                ));
    } catch (Exception e) {
        return Result.failure(e);
    }
}
```

本书附带的代码中提供的基准测试程序允许你测量性能的改善。当然，改善的程度可能会因运行程序的机器而异。

8.5 总结

- 可以通过对记忆化的使用来加速列表的处理。
- 可以将一个 Result 的 List 实例转换为一个 List 的 Result。
- 可以通过压缩来复合两个列表。也可以解压缩列表以生成一个列表 Tuple。
- 可以使用显式递归来实现对列表元素的索引访问。
- 当得到的结果为“零”时，可以执行 foldLeft 的特殊版本来脱离折叠。
- 可以通过展开一个函数和一个终止条件来创建列表。
- 列表可以自动拆分，用以自动地并行处理子列表。

使用惰性

本章要点

- 理解惰性的的重要性
- 在 Java 中实现惰性
- 创建一个惰性的列表数据结构：Stream
- 通过记忆已计算的值来优化惰性列表
- 处理无限流（stream）

有些语言是惰性的，有些不是。这是否意味着有些语言比别的语言付出了更多努力？绝非如此。惰性（laziness）是严格（strictness）的对立面。它与语言是否努力无关，虽然有时你可以认为惰性语言并不要求程序员像使用严格语言那样必须努力地工作。

正如你即将看到的那样，惰性在一些特定问题上具有很多优点，例如复合无限数据结构和计算错误条件等。

9.1 理解严格和惰性

对方法的参数而言，严格意味着参数一旦被方法接收到，便会立即求值。惰性

意味着只有在需要的时候参数才会被求值。

当然，严格和惰性不仅是对于方法的参数而言，而是对所有事情。例如，思考以下声明：

```
int x = 2 + 3;
```

因为 Java 是严格的语言，这里的 x 立即被赋值为 5，求和操作立即进行。让我们看另一个例子：

```
int x = getValue();
```

在 Java 中，在声明变量 x 时，`getValue` 方法会被立即调用，以提供对应的值。另一方面，在惰性语言里，`getValue` 方法只会在即将使用变量 x 时才会被调用。二者之间差异巨大。

例如，请看以下 Java 程序：

```
public static void main(String... args) {  
    int x = getValue();  
}  
  
public static int getValue() {  
    System.out.println("Returning 5");  
    return 5;  
}
```

这段程序将会在控制台上打印 `Returning 5`，因为 `getValue` 方法会被立即调用，虽然返回值不会被用到。在惰性语言里，什么都没有被赋值，所以控制台上什么都不会被打印出来。

9.1.1 Java 是一门严格的语言

Java 原则上没有惰性的顾虑。Java 是严格的。什么东西都是立即被计算的。我们说方法的参数是值传递，即首先计算它们，然后才传入求完了的值。另一方面，在惰性语言中，我们说方法的参数是名称传递，即暂不求值。别被 Java 里方法的参数通常是引用的事情搞糊涂了。引用是地址，这些地址被传入了值。

有些语言是严格的（例如 Java）；有些是惰性的；有些默认是严格的，但是可以选择惰性；有些默认是惰性的，但是也可以选择严格。

然而，Java 并不总是严格的。在 Java 里有一些惰性结构：

- 布尔运算符 `||` 和 `&&`
- 三目运算符 `?:`
- `if...else`
- `for` 循环
- `while` 循环
- Java 8 中的流

如果思考一下，你很快就会意识到，如果 Java 不提供这些惰性，那它就什么也做不了。你能想象一个 `if...else` 结构的两个分支都会系统地被计算吗？或者能想象一个不可能从中脱离的循环吗？所有语言都需要在某种情况下提供惰性。不过，标准 Java 所能提供的惰性对函数式编程而言还不够。

9.1.2 严格带来的问题

严格在像 Java 这样的语言中是如此重要，以至于它被许多程序员当作对表达式赋值的唯一可能性，即便完全严格的语言在现实中什么都做不了。此外，Java 的文档并没有使用 *non-strict*（非严格）或 *lazy*（惰性）这两个词语来描述惰性结构。例如，布尔运算符 `||` 和 `&&` 不叫惰性，而叫 *short-circuiting*（短路）。但简单的事实是：这些运算符对于它们的参数而言是非严格的。我们可以很容易地展示这与方法参数的“严格”求值有何不同。

想象你打算用函数来模拟布尔运算符。清单 9.1 展示了你所要做的。

清单 9.1 `and` 和 `or` 逻辑方法

```
public class BooleanMethods {

    public static void main(String[] args) {
        System.out.println(or(true, true));
        System.out.println(or(true, false));
        System.out.println(or(false, true));
        System.out.println(or(false, false));
        System.out.println(and(true, true));
        System.out.println(and(true, false));
        System.out.println(and(false, true));
        System.out.println(and(false, false));
    }

    public static boolean or(boolean a, boolean b) {
        return a ? true : b ? true : false;
    }
}
```

```

public static boolean and(boolean a, boolean b) {
    return a ? b ? true : false : false;
}
}

```

当然有用布尔运算符的简单办法，但是你的目标是避免使用这些运算符。完成了吗？运行这段程序将会在控制台上显示以下结果：

```

true
true
true
false
true
false
false
false
false

```

到目前还好。但现在请尝试运行清单 9.2 中的程序。

清单 9.2 严格的程序

```

public class BooleanMethods {

    public static void main(String[] args) {
        System.out.println(getFirst() || getSecond());
        System.out.println(or(getFirst(), getSecond()));
    }

    public static boolean getFirst() {
        return true;
    }

    public static boolean getSecond() {
        throw new IllegalStateException();
    }

    public static boolean or(boolean a, boolean b) {
        return a ? true : b ? true : false;
    }

    public static boolean and(boolean a, boolean b) {
        return a ? b ? true : false : false;
    }
}

```

这段程序打印出以下信息：

```

true
Exception in thread "main" java.lang.IllegalStateException

```

显然，`or` 方法与 `||` 运算符并不等价。区别就是 `||` 惰性地进行计算操作数，这意味着当第一个操作数为 `true` 时，第二个操作数并不会被计算，因为它不是计算结果所必需的。但是 `or` 方法严格地计算它的参数，这意味着即使不需要第二个参数的值也会计算它，从而导致抛出 `IllegalStateException`。

当你在第 6 章和第 7 章中使用 `getOrDefault` 方法时也碰到过这个问题，因为参数总是会被赋值，即使计算已经成功。

9.2 实现惰性

惰性在许多场合中都是必需的。Java 其实对像 `if...else`、循环和 `try...catch` 这样的代码块结构使用了惰性。没有惰性的话，一段 `catch` 代码块甚至会在没有异常的时候进行计算。在提供错误行为和操作无限数据结构时必须实现惰性。

在 Java 里实现惰性并不是完全不可能的，你可以通过上一章中用过的 `Supplier` 类来创造出一个相似的：

```
public interface Supplier<T> {  
    T get();  
}
```

请注意你创建了自己的类，但是 Java 8 也提供了一个 `Supplier` 类。究竟使用哪一个你就自己决定吧。它们完全等价。

通过使用 `Supplier` 类，你可以如下重写 `BooleanMethods` 的例子，如清单 9.3 所示。

清单 9.3 使用惰性模拟 Boolean 运算符

```
public class BooleanMethods {  
  
    public static void main(String[] args) {  
        System.out.println(getFirst() || getSecond());  
        System.out.println(or(() -> getFirst(), () -> getSecond()));  
    }  
  
    public static boolean getFirst() {  
        return true;  
    }  
  
    public static boolean getSecond() {  
        throw new IllegalStateException();  
    }  
}
```



```

    }

    public static boolean or(Supplier<Boolean> a, Supplier<Boolean> b) {
        return a.get() ? true : b.get() ? true : false;
    }

    public static boolean and(Supplier<Boolean> a, Supplier<Boolean> b) {
        return a.get() ? b.get() ? true : false : false;
    }
}

```

这段程序打印如下：

```

true
true

```

虽然你不得不改变方法的签名，但是基本上解决了惰性的问题。对使用惰性而言，这个代价并不高。当然，如果参数的求值速度非常快，或者它们已经有值了（例如字面量），这样做多少有点杀鸡用牛刀的感觉。但是如果求值需要耗时很久，这样做可能节省大量的时间。还有，如果求值会有副作用，这样做可能会完全改变程序的结果。

9.3 只有惰性才能做到的事

迄今为止，看起来 Java 在惰性求值上的缺失不是什么大问题。毕竟，如果你可以直接用 Boolean 运算符，为什么还要劳神去重写 Boolean 方法？那是因为惰性还有其他有用的场景。

甚至还有一些不借助惰性就不能实现的算法。我说过，一个严格版的 if...else 一点用都没有。思考以下算法：

1. 接收一个正整型列表。
2. 过滤元素，只保留质数。
3. 返回前十个结果的列表。

这是一个找寻前十个质数的算法，但是此算法无法在没有惰性的情况下实现。如果你不相信，尽管试试看。从第一行开始。如果你是严格的，首先会对正整型列表求值。这样便没有机会去执行第二行，因为这个整型列表是无限的，而你将在到达（不存在的）终点之前耗尽所有的可用内存。

很显然，这个算法无法不通过惰性实现，但是你知道如何用另一种算法来代替

它。先前的算法是函数式的。如果想要找出不借助惰性的结果，那你就不得不将其替换成一个命令式的算法，如下所示：

1. 接收第一个整型。
2. 检查它是否为质数。
3. 如果是，将其保存在一个列表里。
4. 检查结果列表是否有十个元素。
5. 如果有了十个元素，将其作为结果返回。
6. 如果没有，递增整型。
7. 跳到第 2 行继续。

它当然能够工作。但这真是乱七八糟。首先，这是一个糟糕的方式。如果不想测试偶数，你是不是还得把测试的整型加 2 而不是加 1？那为什么又要测试 3、5 等的倍数呢？更重要的是，它并没有表达出问题的本质。这只是计算结果的步骤而已。

这并不是说实现细节（例如不测试偶数）对于获得良好的性能而言不重要。但这些实现细节应该明确地与问题的定义分开。命令式的描述并不是对问题的描述——而是对给出相同结果的另一个问题的描述。

在函数式编程中，一般用一种特殊的结构来解决这个问题：惰性列表，称其为 Stream。

9.4 为何不要用Java 8中的Stream

Java 8 引入了一个被称为 Stream（流）的新结构。你可以用它来进行这种计算吗？嗯，可以，但有几个不要这么做的理由：

- 定义自己的结构更加有益。如果这样做，你会学到和理解许多使用 Java 8 的 Stream 时甚至不会想到的事情。
- Java 的流是一种非常强大的工具，但并不是你所需的工具。Java 8 中的流是出于自动并行化的考虑而设计的。为了允许自动并行化，做了许多妥协。缺失了许多函数式方法，因为它们会使自动并行化变得更加困难。
- Java 8 中的流是有状态的。一旦它们被用于某些操作，这些操作将改变流的状态并使其不再可用。
- 折叠 Java 8 中的流是一种严格的操作，会导致对所有元素求值。

出于所有这些原因，你将在本章中定义自己的流。在完成本章之后，你可能还会喜欢用 Java 8 的流，但你将完全理解 Java 8 的实现中所缺失的内容。

9.5 创建一个惰性列表数据结构

现在，你知道了如何将未赋值的数据表示为 Supplier 实例，你可以轻松地定义一个惰性列表数据结构。它将被称为 Stream，与第 5 章中开发的单链表非常相似，有着一些微妙但极为重要的差异。清单 9.4 展示了 Stream 数据类型的初始定义。

清单 9.4 Stream 数据类型

```
import com.fpinjava.common.Supplier;
```

```
public abstract class Stream<A> {
```

```
    private static Stream EMPTY = new Empty();
```

```
    public abstract A head();
```

```
    public abstract Stream<A> tail();
```

```
    public abstract Boolean isEmpty();
```

```
    private Stream() {}
```

空流被表示为
非参数化单例

Stream 类的构造函数是私有的，
以防止直接被实例化

```
    private static class Empty<A> extends Stream<A> {
```

```
        @Override
```

```
        public Stream<A> tail() {
```

```
            throw new IllegalStateException("tail called on empty");
```

```
        }
```

```
        @Override
```

```
        public A head() {
```

```
            throw new IllegalStateException("head called on empty");
```

```
        }
```

```
        @Override
```

```
        public Boolean isEmpty() {
```

```
            return true;
```

```
        }
```

```
    }
```

Empty 子类与 List.Nil 子类
完全相同

head 并不被
求值，采用
Supplier<T>
的形式

```
    private static class Cons<A> extends Stream<A> {
```

```
        private final Supplier<A> head;
```

```
        private final Supplier<Stream<A>> tail;
```

```
        private Cons(Supplier<A> h, Supplier<Stream<A>> t) {
```

```
            head = h;
```

```
            tail = t;
```

一个非空流由 Stream 子
类表示

类似的，tail 表示
为 <Supplier<T>>，
通过调用相应的
get 方法求值

```

}

@Override
public A head() {
    return head.get();
}

@Override
public Stream<A> tail() {
    return tail.get();
}

@Override
public Boolean isEmpty() {
    return false;
}

static <A> Stream<A> cons(Supplier<A> hd, Supplier<Stream<A>> tl) {

}

@SuppressWarnings("unchecked")
public static <A> Stream<A> empty() {
    return EMPTY;
}

public static Stream<Integer> from(int i) {
    return cons(() -> i, () -> from(i + 1));
}

```

head 方法在返回计算值之前对 head 求值

tail 方法在返回计算值之前对 tail 求值

cons 工厂方法通过调用私有的 Cons 构造函数构造一个 Stream

empty 工厂方法返回 EMPTY 单例

from 工厂方法返回一个始于给定值的无限整型流

以下是使用该 Stream 类型的示例：

```

Stream<Integer> stream = Stream.from(1);
System.out.println(stream.head());
System.out.println(stream.tail().head());
System.out.println(stream.tail().tail().head());

```

这段程序的输出如下：

```

1
2
3

```

这似乎没有什么用。要使 Stream 成为一个有价值的工具，你还需要往里添加一些方法。但首先得把它稍微优化一下。

9.5.1 记忆已计算的值

隐含在惰性之后的思想是，只在需要时才计算数据来节省时间。这意味着你必须在首次访问数据时对其求值。但是，在之后的访问中重新计算它就是浪费时间。由于你正在编写函数式程序，多次求值不会造成什么影响，但是会降低程序的速度。记忆已计算的值是一种解决方案。

为此，你必须在 `Cons` 类中添加字段，用以表示已计算的值：

```
private final Supplier<A> head;
private A h;
private final Supplier<Stream<A>> tail;
private Stream<A> t;
```

然后更改如下取值方法：

```
public A head() {
    if (h == null) {
        h = head.get();
    }
    return h;
}

public Stream<A> tail() {
    if (t == null) {
        t = tail.get();
    }
    return t;
}
```

这种众所周知的技术并不是函数式编程所特有的。它有时被称为按需求值 (evaluation on demand/evaluation as needed)，或惰性求值 (lazy evaluation)。当第一次使用该值时，求值的字段为 `null`，因此会对其求值。在后续访问中，不会再次对其求值，而是返回先前已求好的值。

有些语言默认或可选地提供惰性求值作为标准功能。使用这些语言，你无须再借助空引用和可变字段。不幸的是，Java 并不是这些语言中的一员。在 Java 中，若想稍后再初始化值，最常用的方法是：对于对象类型，先赋以一个空引用；对于原始类型，赋以一个标记值。这样做有风险，因为不能保证在需要该值时，它确实会被初始化为一个有意义的值。`null` 引用可能会导致抛出 `NullPointerException`，它至少还会让人注意到是否已经正确实现了异常处理，但一个零值也许是一个有业务意义的合理值，从而导致程序静默地使用这个可接受但不正确的值。

你也可以使用 `Result<A>` 来表示这个值。这样做能够避免误用 `null` 引用，但你仍然需要使用可变字段。因为所有的这些都是私有的，所以用 `null` 也可以接受。但是如果你愿意，可以使用一个 `Result`（或者一个 `Option`）来表示 `h` 和 `t` 字段。

请注意，虽然 `h` 和 `t` 字段必须是可变的，但它们无须同步。可能发生的最糟糕的事情是：一个线程将检查该字段并发现其为空，接着第二个线程也可能在第一个线程初始化之前检查该字段。最终的结果是，该字段将被初始化两次，存在值不同（就算相等）的可能性。这本身不是一个大问题；写入引用是原子的，所以数据不会被破坏。但是，这可能会导致相应的对象在内存中有两个实例并存。如果你仅验证对象是否相等，那它还不算什么问题，但是如果你验证它们的标识（当然，你永远不会这样做），那就麻烦了。

还要注意，`null` 引用和可变字段是可以完全避免的，只不过要以轻微修改其他地方为代价。尝试搞清楚如何做到这一点。如果你不知道怎么做，先记着这个想法，我们还会在接近本章末尾时再回来。

清单 9.5 展示了完整的 `Stream` 类，它实现了对 `head` 和 `tail` 的惰性求值。

清单 9.5 完整的 `Stream` 类

```
abstract class Stream<A> {  
  
    private static Stream EMPTY = new Empty();  
    public abstract A head();  
    public abstract Stream<A> tail();  
    public abstract Boolean isEmpty();  
    private Stream() {}  
  
    private static class Empty<A> extends Stream<A> {  
  
        @Override  
        public Stream<A> tail() {  
            throw new IllegalStateException("tail called on empty");  
        }  
  
        @Override  
        public A head() {  
            throw new IllegalStateException("head called on empty");  
        }  
  
        @Override  
        public Boolean isEmpty() {  
            return true;  
        }  
    }  
}
```

```

    }

    private static class Cons<A> extends Stream<A> {
        private final Supplier<A> head;
        private A h;
        private final Supplier<Stream<A>> tail;
        private Stream<A> t;

        private Cons(Supplier<A> h, Supplier<Stream<A>> t) {
            head = h;
            tail = t;
        }

        @Override
        public A head() {
            if (h == null) {
                h = head.get();
            }
            return h;
        }

        @Override
        public Stream<A> tail() {
            if (t == null) {
                t = tail.get();
            }
            return t;
        }

        @Override
        public Boolean isEmpty() {
            return false;
        }
    }

    static <A> Stream<A> cons(Supplier<A> hd, Supplier<Stream<A>> tl) {
        return new Cons<>(hd, tl);
    }

    static <A> Stream<A> cons(Supplier<A> hd, Stream<A> tl) {
        return new Cons<>(hd, () -> tl);
    }

    @SuppressWarnings("unchecked")
    public static <A> Stream<A> empty() {
        return EMPTY;
    }

    public static Stream<Integer> from(int i) {
        return cons(() -> i, () -> from(i + 1));
    }
}

```

记忆已计算的 head 的方法

简化创建流的便捷方法

练习 9.1

编写一个 `headOption` 方法，返回流的已计算的 `head`。该方法将在 `Stream` 父类中被声明为如下签名：

```
public abstract Result<A> headOption();
```

答案 9.1

`Empty` 的实现返回一个空 `Result`：

```
@Override
public Result<A> headOption() {
    return Result.empty();
}
```

`Cons` 的实现返回已计算的 `head` 的 `Success`：

```
@Override
public Result<A> headOption() {
    return Result.success(head());
}
```

9.5.2 对流的操作

在本章的剩余部分，你将学习如何在复合流的同时充分利用数据未求值的事实。但是为了查看流，你需要一种方法来求它们的值。对流的所有元素求值可以通过将其转换为 `List` 来完成。或者你可以通过仅计算前 n 个元素或通过在满足条件时对元素求值来处理流。

练习 9.2

创建一个 `toList` 方法将 `Stream` 转换为 `List`。

提示

你可以在 `Stream` 类中实现一个显式递归的方法。

答案 9.2

递归版本将把流的 `head` 简单地 `cons` 到 `toList` 方法应用于 `tail` 的结果上。当然，你需要使这个过程成为尾递归，以便能够通过 `TailCall` 得到一个栈安全的实现：


```

public List<A> toList() {
    return toList(this, List.list()).eval().reverse();
}

private TailCall<List<A>> toList(Stream<A> s, List<A> acc) {
    return s.isEmpty()
        ? ret(acc)
        : sus(() -> toList(s.tail(), List.cons(s.head(), acc)));
}

```

请注意，此处未显示 `TailCall.ret()` 和 `TailCall.sus()` 的静态导入。

请注意，在如 `Stream.from(1)` 创建的无限流上调用 `toList` 会创建无限列表。与流不同，该列表被及早求值，所以理论上它会导致一个永不停止的程序。（实际上，它将以 `OutOfMemoryError` 告终。）请确保在运行程序之前创建一个将会截断列表的条件，如练习 9.3 所示。

练习 9.3

编写一个 `take(n)` 方法以返回流的前 n 个元素，还有在删除前 n 个元素后返回剩余流的 `drop(n)` 方法。请注意，你必须确保在调用这些方法时不会求值。以下是它在 `Stream` 父类中的签名：

```

public abstract Stream<A> take(int n);
public abstract Stream<A> drop(int n);

```

答案 9.3

`Empty` 类中的两个实现都返回 `this`。对于 `Cons` 类中的 `take` 方法，你需要通过传入流的未求值 `head`（意味着引用 `head` 字段而不是调用 `head()` 方法）来调用 `cons` 方法以创建一个新的 `Stream<A>`，并对流的 `tail` 递归调用 `take(n - 1)` 直到 $n == 1$ 为止。`drop` 方法更简单。你只需在 $n > 0$ 时递归地对流的 `tail` 调用 `drop(n - 1)` 即可。请注意，无须使 `take` 方法栈安全，因为对 `take` 的递归调用已经是惰性的了。

```

public Stream<A> take(int n) {
    return n <= 0
        ? empty()
        : cons(head, () -> tail().take(n - 1));
}

```

`take` 方法允许你用截取无限流中一定数量的元素的方式来使用它。但请注意，

在将其转换为列表之前，必须在流上调用该方法：

```
List<Integer> list = Stream.from(1).take(10).toList();
```

在结果列表上调用等价的方法，将会挂起直至内存耗尽，导致 `OutOfMemoryError`：

```
List<Integer> list = Stream.from(1).toList().takeAtMost(10);
```

与此相反，`drop` 方法必须是栈安全的：

```
public Stream<A> drop(int n) {
    return drop(this, n).eval();
}

public TailCall<Stream<A>> drop(Stream<A> acc, int n) {
    return n <= 0
        ? ret(acc)
        : sus(() -> drop(acc.tail(), n - 1));
}
```

练习 9.4

编写一个 `takeWhile` 方法，只要条件匹配，它将返回包含所有起始元素的 `Stream`。`Stream` 父类中的方法签名如下：

```
public abstract Stream<A> takeWhile(Function<A, Boolean> p)
```

提示

请注意，与 `take` 和 `drop` 不同，这个方法将会对元素求值，因为它必须检查第一个元素以验证其是否满足断言所表示的条件。你应该核实它是否只计算了流的第一个元素。

答案 9.4

这个方法与 `take` 方法非常相似。主要区别在于终止条件不再是 `n <= 0`，而是提供的函数返回 `false`：

```
public Stream<A> takeWhile(Function<A, Boolean> f) {
    return f.apply(head())
        ? cons(head, () -> tail().takeWhile(f))
        : empty();
}
```

```

public List<A> toList() {
    return toList(this, List.list()).eval().reverse();
}

private TailCall<List<A>> toList(Stream<A> s, List<A> acc) {
    return s.isEmpty()
        ? ret(acc)
        : sus(() -> toList(s.tail(), List.cons(s.head(), acc)));
}

```

请注意，此处未显示 `TailCall.ret()` 和 `TailCall.sus()` 的静态导入。

请注意，在如 `Stream.from(1)` 创建的无限流上调用 `toList` 会创建无限列表。与流不同，该列表被及早求值，所以理论上它会导致一个永不停止的程序。（实际上，它将以 `OutOfMemoryError` 告终。）请确保在运行程序之前创建一个将会截断列表的条件，如练习 9.3 所示。

练习 9.3

编写一个 `take(n)` 方法以返回流的前 n 个元素，还有在删除前 n 个元素后返回剩余流的 `drop(n)` 方法。请注意，你必须确保在调用这些方法时不会求值。以下是它在 `Stream` 父类中的签名：

```

public abstract Stream<A> take(int n);
public abstract Stream<A> drop(int n);

```

答案 9.3

`Empty` 类中的两个实现都返回 `this`。对于 `Cons` 类中的 `take` 方法，你需要通过传入流的未求值 `head`（意味着引用 `head` 字段而不是调用 `head()` 方法）来调用 `cons` 方法以创建一个新的 `Stream<A>`，并对流的 `tail` 递归调用 `take(n - 1)` 直到 $n == 1$ 为止。`drop` 方法更简单。你只需在 $n > 0$ 时递归地对流的 `tail` 调用 `drop(n - 1)` 即可。请注意，无须使 `take` 方法栈安全，因为对 `take` 的递归调用已经是惰性的了。

```

public Stream<A> take(int n) {
    return n <= 0
        ? empty()
        : cons(head, () -> tail().take(n - 1));
}

```

`take` 方法允许你用截取无限流中一定数量的元素的方式来使用它。但请注意，

在将其转换为列表之前，必须在流上调用该方法：

```
List<Integer> list = Stream.from(1).take(10).toList();
```

在结果列表上调用等价的方法，将会挂起直至内存耗尽，导致 `OutOfMemoryError`：

```
List<Integer> list = Stream.from(1).toList().takeAtMost(10);
```

与此相反，`drop` 方法必须是栈安全的：

```
public Stream<A> drop(int n) {
    return drop(this, n).eval();
}

public TailCall<Stream<A>> drop(Stream<A> acc, int n) {
    return n <= 0
        ? ret(acc)
        : sus(() -> drop(acc.tail(), n - 1));
}
```

练习 9.4

编写一个 `takeWhile` 方法，只要条件匹配，它将返回包含所有起始元素的 `Stream`。`Stream` 父类中的方法签名如下：

```
public abstract Stream<A> takeWhile(Function<A, Boolean> p)
```

提示

请注意，与 `take` 和 `drop` 不同，这个方法将会对元素求值，因为它必须检查第一个元素以验证其是否满足断言所表示的条件。你应该核实它是否只计算了流的第一个元素。

答案 9.4

这个方法与 `take` 方法非常相似。主要区别在于终止条件不再是 `n <= 0`，而是提供的函数返回 `false`：

```
public Stream<A> takeWhile(Function<A, Boolean> f) {
    return f.apply(head())
        ? cons(head(), () -> tail().takeWhile(f))
        : empty();
}
```

由于并未计算递归调用，所以你还是不必使这个方法栈安全。Empty 的实现返回 this。

练习 9.5

编写一个 dropWhile 方法，返回一个前面那些满足条件的元素被删除的流。Stream 父类中的签名如下：

```
public Stream<A> dropWhile(Function<A, Boolean> p);
```

提示

你需要编写这个方法的尾递归版本才能使其实现栈安全。

答案 9.5

与之前的递归方法一样，解决方案将包含一个主方法，它会调用栈安全的辅助递归方法并对其结果求值：

```
public Stream<A> dropWhile(Function<A, Boolean> p) {
    return dropWhile(this, p).eval();
}

private TailCall<Stream<A>> dropWhile(Stream<A> acc,
                                       Function<A, Boolean> p) {
    return acc.isEmpty()
        ? ret(acc)
        : p.apply(acc.head())
            ? sus(() -> dropWhile(acc.tail(), p))
            : ret(acc);
}
```

由于这个方法用了一个辅助方法，它可以在 Stream 父类中实现。

9.6 惰性的真正本质

惰性经常被认为是只有在（如果）需要时才对表达式求值。其实这只是惰性的一个应用。

惰性到底意味着什么

严格性和惰性之间的真正区别在于：严格性与做事有关，而惰性则与指出要

做的事有关。数据的惰性求值指出数据必须在将来的某个时间被计算。但惰性并不仅限于计算数据。

在Java中打印到控制台是严格的，因为它是一个作用，所以与函数式编程不兼容。但是指出你以后应该打印到控制台（可以被称为“惰性打印”）就不一样了。这种惰性的作用只是生成数据，可以作为程序的结果返回。第13章有更多关于这个主题的讨论。

以一个非常简单的命令式程序为例：

```
List<String> names = ...
for(String name : names) {
    System.out.println(String.format("Hello, %s!", name));
}
```

该程序适用于严格性，因为它会对列表中的每个名称都执行该做的事。该程序的惰性版本可能如下所示：

```
List<String> names = ...
names.map(name -> (Runnable) () -> System.out.println(name));
```

该程序并不打印每个名称，而是生成用于打印名称的指令列表。换句话说，这段程序编写了一个可以稍后执行的程序。重要的是理解两个程序并不等价，因为运行它们并不会生成相同的结果。但是第二个程序的输出等价于第一个程序本身，因为如果运行第二个程序的输出，将得到与运行第一个程序完全相同的结果。

当然，为了运行第二个程序的输出，你需要一些解释器。你将在第13章中学习如何做到这一点（尽管你可能已经对所涉及的内容有了一个好主意）。

这种办法的一个巨大优点是你可以生成一个生成错误的程序的描述，然后根据某些条件决定不执行。或者你也可以生成一个无限的表达式，然后采取一些手段将它化简为一个有限的表达式。

当编写一个方法来模拟布尔运算符的惰性时，你已经看到了第一种情况的示例。对于第二种情况而言，假设你有一个所有正整数的列表。在命令式编程中，可以如下编写：

```
for (int i = 0;; i++) {}
```

尽管什么事情都不做，这样的程序也永远不会终止。但是，如果要找到斐波那

取值大于 500 的第一个整数，你可以如下编写：

```
for (int i = 0;; i++) {
    if (fibonacci(i) > 500) return i;
}
```

现在你的程序终止了，因为在找到答案后，整数列表将停止计算。这是因为 for 循环是一个惰性的结构。虽然 `for (int i = 0;; i++)` 表示无限序列的整数，但它只会按需求值。

在第 8 章中，你在 `List` 类中创建了如下 `exists` 方法：

```
public Boolean exists(Function<T, Boolean> p) {
    return p.apply(head()) || tail().exists(p);
}
```

该方法遍历列表直到找到一个满足断言 `p` 的元素为止。不会再检查列表的剩余部分了，因为 `||` 运算符是惰性的，如果第一个参数值为 `true`，则不会计算第二个参数。

练习 9.6

为 `Stream` 创建一个 `exists` 方法。该方法应该对元素求值，直到满足条件为止。如果没有满足条件，则将对所有元素求值。

答案 9.6

一个简单的方案可能与 `List` 的 `exists` 方法非常相似：

```
public boolean exists(Function<A, Boolean> p) {
    return p.apply(head()) || tail().exists(p);
}
```

当然，你应该令其栈安全。为了编写一个栈安全的实现，你必须首先使其成为尾递归，然后使用 `TailCall` 类：

```
public boolean exists(Function<A, Boolean> p) {
    return exists(this, p).eval();
}

private TailCall<Boolean> exists(Stream<A> s, Function<A, Boolean> p) {
    return s.isEmpty()
        ? ret(false)
        : p.apply(s.head())
            ? ret(true)
```

```

        : sus(() -> exists(s.tail(), p));
    }

```

此版本适用于两个子类，因此可以将其放在 `Stream` 父类中。

9.6.1 折叠流

在第 5 章中，你看到了如何将递归抽象为折叠方法，你还学习了如何将列表向右或向左折叠。折叠流有点不同。虽然原理相同，但主要的区别是流未被求值。递归操作可能会栈溢出并导致抛出一个 `StackOverflowException`，但递归操作的描述则不会。结果就是在 `List` 中不能栈安全的 `foldRight` 在绝大多数情况下都不会栈溢出。如果要对每个操作求值，例如对 `Stream<Integer>` 的元素求和，那它将会溢出；但如果不是，那么它将构造一个未计算的操作描述，而不是对操作求值。

另一方面，基于 `foldLeft`（可以使之栈安全）的 `foldRight` 的 `List` 实现不能与流一起使用，因为它需要反转流，而这将导致对所有的元素求值；在无限流的情况下它甚至不可能做到。由于反转了计算的方向，所以 `foldLeft` 的栈安全版本也不再能用了。

练习 9.7

为流创建一个 `foldRight` 方法。此方法将类似于 `List.foldRight` 方法，但你应该关注惰性。

提示

惰性表示为 `Supplier<T>` 元素而非 `T`。`Stream` 父类中的方法签名为

```

public abstract <B> B foldRight(Supplier<B> z,
                                Function<A, Function<Supplier<B>, B>> f);

```

答案 9.7

`Empty` 类的实现显而易见：

```

public <B> B foldRight(Supplier<B> z,
                      Function<A, Function<Supplier<B>, B>> f) {
    return z.get();
}

```

以下是 `Cons` 的实现：


```
public <B> B foldRight(Supplier<B> z,
    Function<A, Function<Supplier<B>, B>> f) {
    return f.apply(head()).apply(() -> tail().foldRight(z, f));
}
```

请注意，这个方法不是栈安全的，因此不应该把这类计算用于对大于 1000 个整数的列表求和。尽管如此，你会看到它有许多有趣的用例。

练习 9.8

通过 `foldRight` 实现 `takeWhile` 方法。验证它在长列表上的表现。

答案 9.8

起始值是一个空流的 `Supplier`。它可以写成 `() -> empty()`，不过你也可以用方法引用版的 `Stream::empty`。该函数检查当前元素 (`f.apply(a)`)。如果结果为 `true`（即该元素满足断言 `p` 所表示的条件），则返回 `a` 的 `Supplier` 与当前的流 `cons` 的结果。

```
public Stream<A> takeWhile(Function<A, Boolean> p) {
    return foldRight(Stream::empty, a -> b -> p.apply(a)
        ? cons(() -> a, b)
        : empty());
}
```

你可以通过运行本书附带的代码 (<https://github.com/fpinjava/fpinjava>) 中提供的测试来验证，即使是超过 100 万个元素的数据流，该方法也不会栈溢出。这是因为 `foldRight` 本身并不计算结果。求值取决于用于折叠的函数。如果这个函数构造了一个新流（如在 `takeWhile` 的情况下），则不会对该流求值。

练习 9.9

通过 `foldRight` 实现 `headOption`。

答案 9.9

起始元素是一个未求值的空流 (`Result::empty` 或 `() -> Result.empty()`)。如果流为空，那这就是返回值。用于折叠流的函数将简单地忽略第二个参数，所以在第一次应用（对 `head` 元素）时，它返回 `Result.success(a)`，并且这个结果永远不会改变。

```
public Result<A> headOptionViaFoldRight() {
```

```
return foldRight(Result::empty, a -> ignore -> Result.success(a));
}
```

练习 9.10

通过 `foldRight` 实现 `map`。验证这个方法不会对任何流元素求值。

答案 9.10

从一个空流的 `Supplier` 开始。用于创建折叠的函数将会 `cons` 一个函数对当前元素的未求值应用与当前结果。

```
public <B> Stream<B> map(Function<A, B> f) {
    return foldRight(Stream::empty, a -> b -> cons(() -> f.apply(a), b));
}
```

练习 9.11

通过 `foldRight` 实现 `filter`。验证此方法不会对比所需更多的流元素求值。

答案 9.11

又是从一个未求值的空流开始。用于折叠的函数将待筛选元素应用于当前参数。如果结果为 `true`，则该元素将会用于通过 `cons` 其与当前流的结果来创建新流。否则，当前流的结果保持不变。（在 `b` 上调用 `get` 不会对任何元素求值。）

```
public Stream<A> filter(Function<A, Boolean> p) {
    return foldRight(Stream::empty, a -> b -> p.apply(a)
        ? cons(() -> a, b)
        : b.get());
}
```

请注意，此方法将对流元素求值，直到找到第一个匹配。更多详细信息，请参阅随书附带代码中的相应测试。

练习 9.12

通过 `foldRight` 实现 `append`。在 `append` 方法的参数中应该是非严格的。

答案 9.12

起始元素是你要追加的（未求值的）流。折叠函数简单地通过 `cons` 当前的元素与当前的结果来创建一个新的流。

```
public Stream<A> append(Supplier<Stream<A>> s) {
```

```
return foldRight(s, a -> b -> cons(() -> a, b));
}
```

练习 9.13

通过 `foldRight` 实现 `flatMap`。

答案 9.13

你还是从一个未求值的空流开始。该函数应用于当前元素，生成一个流并往里追加当前的结果。这具有展平结果的作用（将 `Stream<Stream>` 转换为 `Stream`）。

```
public <B> Stream<B> flatMap(Function<A, Stream<B>> f) {
    return foldRight(Stream::empty, a -> b -> f.apply(a).append(b));
}
```

追踪求值与函数应用

注意惰性的后果很重要。如果使用像列表那样的严格集合，连续应用 `map`、`filter` 和一个新 `map` 即意味着遍历列表三次：

```
private static Function<Integer, Integer> f = x -> {
    System.out.println("Mapping " + x);
    return x * 3;
};

private static Function<Integer, Boolean> p = x -> {
    System.out.println("Filtering " + x);
    return x % 2 == 0;
};

public static void main(String... args) {
    List<Integer> list = List.list(1, 2, 3, 4, 5).map(f).filter(p);
    System.out.println(list);
}
```

如你所见，函数 `f` 和 `p` 不是真正的函数，因为它们输出到了控制台。这不是函数式风格，但它有助于你了解发生了什么。你可以通过返回结果和输出字符串列表的元组来轻松实现这个测试的函数式版本。（如果你愿意，可以把它当成一道附加题）。这个程序的输出如下：

```
Mapping 5
Mapping 4
Mapping 3
```

```
Mapping 2
Mapping 1
Filtering 15
Filtering 12
Filtering 9
Filtering 6
Filtering 3
[6, 12, NIL]
```

这表明函数 *f* 处理了所有的元素，即完全遍历列表。接下来，函数 *p* 又处理了所有元素，即对第一个 *map* 的结果列表的第二次完全遍历。

相比之下，看看以下使用 *Stream* 而不是 *List* 的程序：

```
private static Stream<Integer> stream =
    Stream.cons(() -> 1,
        Stream.cons(() -> 2,
            Stream.cons(() -> 3,
                Stream.cons(() -> 4,
                    Stream.cons(() -> 5, Stream.<Integer>empty())))))

private static Function<Integer, Integer> f = x -> {
    System.out.println("Mapping " + x);
    return x * 3;
};

private static Function<Integer, Boolean> p = x -> {
    System.out.println("Filtering " + x);
    return x % 2 == 0;
};

public static void main(String... args) {
    Stream<Integer> result = stream.map(f).filter(p);
    System.out.println(result.toList());
}
```

输出如下：

```
Mapping 1
Filtering 3
Mapping 2
Filtering 6
Mapping 3
Filtering 9
Mapping 4
Filtering 12
Mapping 5
Filtering 15
[6, 12, NIL]
```

你可以看到流的遍历只发生了一次。首先，`f` 映射了元素 1，得到 3。然后 3 被过滤掉（并且因为不是偶数而被舍弃）。接下来 `f` 映射了 2，得到 6，被过滤并作为结果保存起来。

你可以看到，流的惰性允许你复合计算的描述，而非结果。请注意，求值的元素数量降至最低。

如果你使用尚未计算的值来构建流，并在删除输出结果时使用带日志记录的计算方法，则会得到以下结果：

```
Evaluating 1  
Mapping 1  
Filtering 3  
Evaluating 2  
Mapping 2  
Filtering 6
```

你可以看到只有前两个元素被求值。其余的计算是最终打印的结果。

练习 9.14

编写一个 `find` 方法，以断言（一个从 `A` 到 `Boolean` 的函数）为参数并返回一个 `Result<A>`。如果找到了一个匹配断言的元素，则返回结果为 `Success`，否则为 `Empty`。

提示

你几乎没有什么可写的，只需复合前面几节中你写过的两个方法。

答案 9.14

只需复合 `filter` 方法与 `headOption` 方法：

```
public Result<A> find(Function<A, Boolean> p) {  
    return filter(p).headOption();  
}
```

9.7 处理无限流

由于未对流求值，因此它可以是无限的，同时仍然可以在计算中复合。你已见过的 `from` 方法是一个简单的例子：

```
public static Stream<Integer> from(int i) {
```

```
return cons(() -> i, () -> from(i + 1));
}
```

该方法返回一个无限的整型流，从 *i* 开始对每个新元素加 1。这是一个创建有限自增整型流的非常方便的方法：

```
Stream<Integer> stream = from(0).take(10000);
```

这段代码将创建一个 10 000 个整数的流，从 0~9999，而没有任何求值。

练习 9.15

编写一个 `repeat` 方法，它以一个对象为参数并返回同一对象的无限流。

答案 9.15

这个方法与 `from` 方法非常相似：

```
public static <A> Stream<A> repeat(A a) {
    return cons(() -> a, () -> repeat(a));
}
```

练习 9.16

编写一个 `iterate` 方法来泛化 `from` 和 `repeat` 方法，`iterate` 方法具有两个参数：用于第一个值的 `seed` 和计算下一个值的函数。它的签名如下：

```
public static <A> Stream<A> iterate(A seed, Function<A, A> f)
```

然后基于 `iterate` 重写 `from` 和 `repeat` 方法。

答案 9.16

`iterate` 方法具有与 `from` 和 `repeat` 完全相同的结构，不同之处在于初始值和函数已被参数化：

```
public static <A> Stream<A> iterate(A seed, Function<A, A> f) {
    return cons(() -> seed, () -> iterate(f.apply(seed), f));
}
```

```
public static <A> Stream<A> repeat(A a) {
    return iterate(a, x -> x);
}
```

```
public static Stream<Integer> from(int i) {
    return iterate(i, x -> x + 1);
}
```

请注意，由于 `seed` 作为方法参数被传递，因此它在用于创建“未求过的”值 (`Supplier`) 之前已经求过值了。当然，很容易就能创建一个接收未求过值的 `seed` 的 `iterate` 版：

```
public static <A> Stream<A> iterate(Supplier<A> seed, Function<A, A> f) {
    return cons(seed, () -> iterate(f.apply(seed.get()), f));
}
```

练习 9.17

编写一个 `fibs` 函数，生成斐波那契数列的无限流：0、1、1、2、3、5、8 等。

提示

考虑使用 `iterate` 方法生成整型元组的中间流。

答案 9.17

答案在于创建一个元组 (x, y) ，其中 x 和 y 是两个连续的斐波那契数字。一旦生成了这个流，你只需要 `map` 一个从元组到其第一个元素的函数：

```
public static Stream<Integer> fibs() {
    return iterate(new Tuple<>(0, 1),
        x -> new Tuple<>(x._2, x._1 + x._2)).map(x -> x._1);
}
```

练习 9.18

`iterate` 方法可以进一步被泛化。编写一个 `unfold` 方法，以一个起始状态 S 类型和一个从 S 到 `Result<Tuple<A, S>>` 的函数为参数并返回 A 的流。返回 `Result` 可以指明应该停止还是应该继续这个流。

使用状态 S 意味着数据生成的源不必与生成数据的类型相同。要应用这个新方法，可以使用 `unfold` 方法来编写新版本的 `fibs`。`unfold` 的签名如下：

```
public static <A, S> Stream<A> unfold(S z,
    Function<S, Result<Tuple<A, S>>> f)
```

答案 9.18

首先，将函数 f 应用于初始状态 z 。这样会生成一个 `Result<Tuple<A, S>>`。接下来将该结果 `map` 到一个接收 `Tuple<A, S>` 的函数，通过 `cons` 元组的左侧成员 (A 值) 和一个对 `unfold` 并使用右侧成员作为初始状态的 (未求值的)

递归调用来生成一个流。`map` 的结果是 `Success(stream)` 或者 `Empty`。然后使用 `getOrElse` 返回其包含的流或一个默认的空流：

```
public static <A, S> Stream<A> unfold(S z,
                                     Function<S, Result<Tuple<A, S>>> f) {
    return f.apply(z).map(x -> cons(() -> x._1,
                                     () -> unfold(x._2, f))).getOrElse(empty());
}
```

新版 `from` 用整型 `seed` 作为初始状态，还有一个从 `Integer` 到 `Tuple<Integer, Integer>` 的函数。这里的状态与值的类型相同：

```
public static Stream<Integer> from(int n) {
    return unfold(n, x -> Result.success(new Tuple<>(x, x + 1)));
}
```

`fibs` 方法更全面地使用了 `unfold` 方法。状态是一个 `Tuple<Integer, Integer>`，而函数生成的是一个 `Tuple<Integer, Tuple<Integer, Integer>>`：

```
public static Stream<Integer> fibs() {
    return unfold(new Tuple<>(1, 1),
                   x -> Result.success(new Tuple<>(x._1, new Tuple<>(x._2, x._1 + x._2))));
}
```

你可以看到这些方法的实现是多么紧凑和优雅！

9.8 避免null引用和可变字段

在 9.5.1 节中，我说过无须使用 `null` 引用和可变字段就可以很容易地修改你的 `Stream` 类来记忆 `head` 和 `tail`。你找到办法了吗？其实 `tail` 引用的记忆化并不真的有必要，因为 `tail` 本身是一个惰性的结构（一个 `Stream`），所以计算引用不会花费明显的时间。你只记忆 `head` 即可。

避免 `null` 引用很简单：只要尚未求值，你就可以用 `Result.Empty` 来取代 `null`，并使用 `Result.Success` 来保存计算值。为了避免使用可变字段，需要在求值时生成一个新的 `Stream`。为此，你会使用两个构造函数：一个接收未求值的 `head`，另一个接收已求值的 `head`：


```

private final Supplier<A> head;
private final Result<A> h;
private final Supplier<Stream<A>> tail;

private Cons(Supplier<A> h, Supplier<Stream<A>> t) {
    head = h;
    tail = t;
    this.h = Result.empty();
}

private Cons(A h, Supplier<Stream<A>> t) {
    head = () -> h;
    tail = t;
    this.h = Result.success(h);
}

```

因为求值发生在 head 方法中，所以你需要一个新的实现。但你还得返回带有 head 值的新 Stream。你可以让 head 方法返回一个 Tuple<A, Stream<A>>：

```

public Tuple<A, Stream<A>> head() {
    A a = h.getOrElse(head.get());
    return h.isEmpty()
        ? new Tuple<>(a, new Cons<>(a, tail))
        : new Tuple<>(a, this);
}

```

当然，使用 head() 的所有方法现在都必须改为使用 head()._1。而且，如果保持了一个对流的引用，则必须用新流 (head()._2) 来代替。请注意，到目前为止，Stream 类中还没这么做过！

还需要修改 headOption 方法以返回一个元组。你可以在本书附带的代码 (<https://github.com/fpinjava/fpinjava>) 中的 listing09_06 包中找到完整的 Stream 类。

练习 9.19

用 foldRight 来实现各种方法是一种巧妙的技术。不幸的是，它并不适用于 filter。如果你使用不匹配超过 1000 个或 2000 个连续元素的断言来检查该方法，就会导致栈溢出。用新的 Stream 类但不用 null 或可变字段，来编写一个栈安全的 filter 方法。

提示

问题出自让断言返回 false 的长区间元素。尝试想办法摆脱这些元素。

答案 9.19

解决办法是删除使用让 `dropWhile` 方法返回 `false` 的长区间元素。为此，你需要反转条件 (`!p.apply(x)`)，然后检查生成的流是否为空。如果流为空，则将其返回。（任何空流都行，因为空流是一个单例。只要类型正确即可。）如果流非空，则通过 `cons` 流的 `head` 和已过滤的 `tail` 来创建一个新流。

请注意，`head` 方法返回一个元组，因此你必须使用该元组的左侧（第一个）元素作为流的 `head` 元素。理论上，你应该使用元组的右侧（第二个）元素进行后续访问。不这样做会导致对 `head` 重新求值。但是由于第二次只访问了 `tail` 而并不访问 `head`，因此你可以使用 `stream.getTail()` 来代替。这么做允许你避免使用局部变量来引用 `stream.head()` 的结果。

```
public Stream<A> filter(Function<A, Boolean> p) {
    Stream<A> stream = this.dropWhile(x -> !p.apply(x));
    return stream.isEmpty()
        ? stream
        : cons(() -> stream.head()._1,
              () -> stream.tail().filter(p));
}
```

另一种选择是使用 `headOption` 方法。该方法返回一个保存 `Result<A>` 的 `Tuple`，可以通过 `map` 来递归调用以生成新流。最后，这样会生成一个 `Result<Stream<A>>`，如果没有元素满足断言，则其为空。还要完成的全部是在此 `Result` 上调用 `getOrElse`，并传递一个空流作为默认值。

```
public Stream<A> filter(Function<A, Boolean> p) {
    Stream<A> stream = this.dropWhile(x -> !p.apply(x));
    return stream.headOption()._1.map(a -> cons(() -> a,
        () -> stream.tail().filter(p))).getOrElse(empty());
}
```

9.9 总结

- 严格求值意味着一旦被引用就会求值。
- 惰性求值意味着只有在需要时才会求值。
- 有些语言是严格的，有些是惰性的。有些是默认的惰性和可选的严格，有些是默认的严格和可选的惰性。

- Java 是一门严格的语言。对于方法的参数而言是严格的。
- 虽然 Java 并非惰性的，但是可以使用 Supplier 接口实现惰性。
- 惰性允许操纵和复合无限数据结构。
- Stream 并未求值，它可能是无限的列表。
- 可以使用记忆化来避免多次求相同的值。
- 右折叠不会导致流求值。只有一些用于折叠的函数才会。
- 可以用折叠复合多个迭代操作，而不会导致多次迭代。
- 可以很容易定义和复合无限流。

10

用树进行更多数据处理

本章要点

- 了解树结构中大小、高度和深度之间的关系
- 了解插入顺序和二叉搜索树的结构之间的关系
- 遍历树的各种顺序
- 实现二叉搜索树
- 合并、折叠和平衡树

在第 5 章中，你学到的单链表可能是函数式编程中使用最广的数据结构。尽管列表是对许多操作而言是效率非常高的数据结构，但是它也有一些局限性，主要是访问元素的复杂性与元素的数量成正比。例如，如果要搜索的元素是列表的最后一个元素，可能需要检查所有的元素才能找到它。还有其他效率较低的操作，例如排序、通过索引访问元素，还有找到最大或最小的元素。显然，要找到列表中的最大（或最小）元素，必须遍历整个列表。在本章中，你将学到可以解决这些问题的数据结构：二叉树。

10.1 二叉树

数据树是与列表不同的结构，它的每个元素都会链接不止一个元素。在某些树中，一个元素（有时被称为节点）可能链接了数量不等的其他元素。然而最常见的还是元素链接了固定数量的元素。在二叉树中也一样，每个元素都链接了两个元素。这些链接被称为分支。在二叉树中，我们说左右分支。图 10.1 展示了一个二叉树的示例。

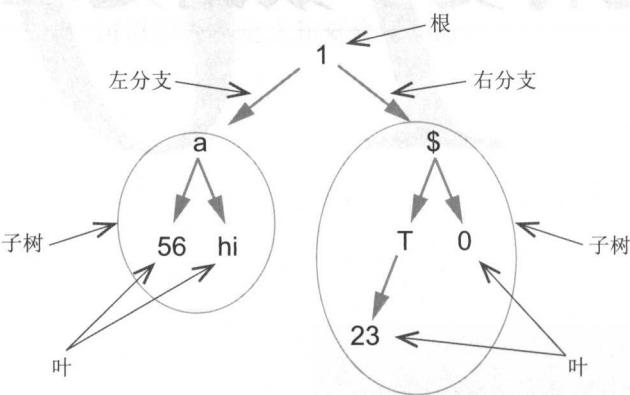


图 10.1 二叉树是由一个根和两个分支组成的递归结构。左分支是一个指向左子树的链接，右分支是一个指向右子树的链接。被称为叶的终端元素具有空分支（图中未标出）。

图 10.1 中所示的树并不常见，因为它的元素类型不同。换句话说，它是一棵对象树。大多数情况下，你会处理类型更具体的树，例如整型树。在图 10.1 中，你可以看到树是递归结构。每个分支指向一棵新树（有时被称为子树）。你还可以看到一些分支指向了一个单独的元素。这不是一个问题，因为单独的元素实际上是一棵带有空分支的树。还要注意 T 元素：它有一个左分支，但没有右分支。

由此可以推断出二叉树的定义。树具有以下特征之一：

- 单独的元素
- 具有一个分支（右或左）的元素
- 具有两个分支（右和左）的元素

每个分支都拥有一棵(子)树。所有的元素都有两个或零个分支的树称为完整树。图 10.1 中的树并不完整，但其左子树是完整的。

10.1.1 平衡树和非平衡树

二叉树可以有不同程度的平衡。对于一棵完全平衡树而言，其所有子树的两个分支都包含了相同数量的元素。图 10.2 展示了树的三个示例，它们都具有相同的元素。第一棵树是完全平衡的，最后一棵树是完全不平衡的。完全的平衡二叉树有时被称为完美树（perfect tree）。

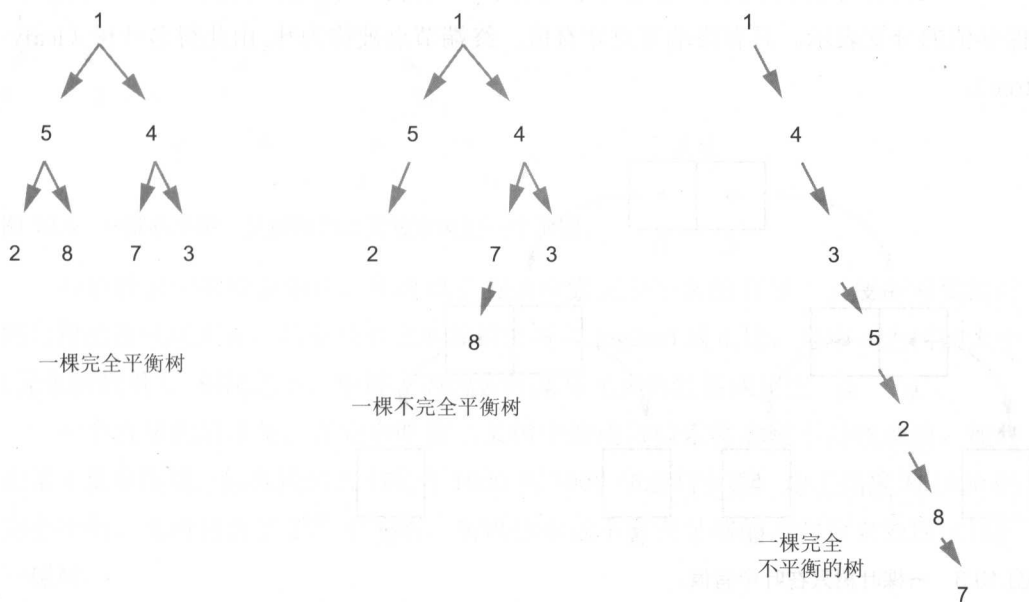


图 10.2 树可以有不同程度的平衡。

在图 10.2 中，右边的树实际上是一个单链表。单链表可以当作完全不平衡的树的一个特殊情况。

10.1.2 大小、高度和深度

树可以通过其包含的元素数量和这些元素所在的层级来描述。元素数量被称为大小，不算根在内的层级被称为高度。在图 10.2 中，所有三棵树的大小均为 7。第一（完全平衡）棵树的高度为 2，第二棵的高度为 3，第三棵的高度为 6。

高度这个词也用于描述单个元素，它指的是从元素到叶的最长路径的长度。根的高度就是树的高度，元素的高度是以此元素为根的子树的高度。

元素的深度是从根到元素的路径的长度。第一个元素也被称为根，其深度为 0。

在图 10.2 所示的完全平衡树中，5 和 4 的深度为 1；而 2、8、7 和 3 的深度为 2。

按照约定，空树的高度和深度等于 -1。你会看到这对于诸如平衡等某些操作而言是必需的。

10.1.3 叶树

二叉树有时被表示为不同的形式，如图 10.3 所示。在这种表示形式中，树由不保存值的分支表示。只有终端节点才有值。终端节点被称为叶，由此得名叶树（leafy tree）。

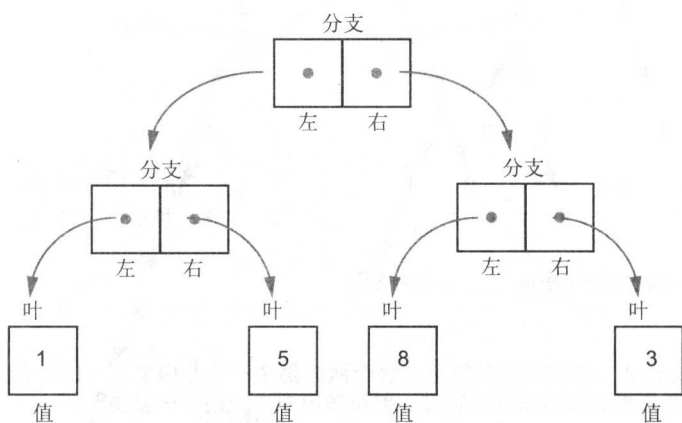


图 10.3 一棵叶树只在叶中有值。

有时会倾向于叶树的表示形式，因为它让一些函数更容易实现。在本书中，我们只会顾及“经典”的树，而非叶树。

10.1.4 有序二叉树或二叉搜索树

一棵有序二叉树，也被称为二叉搜索树（BST），是一种包含可排序元素的树，其中一个分支中所有元素的值都低于根元素，而另一个分支中的所有元素的值都高于根元素。相同的条件也适用于所有子树。根据约定，元素的值低于根的元素位于左分支，而元素的值高于根的元素位于右分支。图 10.4 展示了一个有序树的例子。

有序二叉树定义的一个重要结果就是它们永远也不会有重复的值。

有序树特别有意思，因为它们允许快速检索元素。要确定树中是否包含元素，请遵循以下步骤：

- 1 比较根与要检索的元素的值。如果它们相等，那你就完成了任务。
- 2 如果要检索的元素的值小于根的值，则递归左分支。
- 3 如果要检索的元素的值大于根的值，则递归右分支。

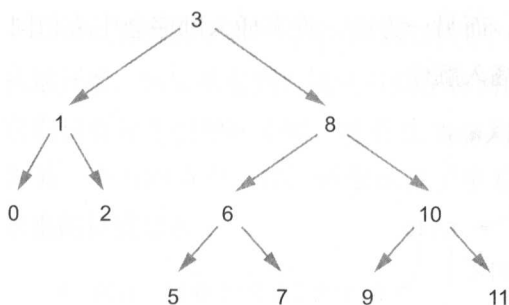


图 10.4 一棵有序树，又被称为二叉检索树的一个示例。

与单链表中的检索相比，你可以看到，检索完全平衡的有序二叉树所需要的时间与树的高度成正比，这意味着它所需的时间与 $\log_2(n)$ 成正比，其中 n 为树的大小（元素的数量）。相比之下，单链表的检索时间与元素的数量成正比。

一个直接的后果是，在完全平衡二叉树中的递归检索将永远不会栈溢出。如你在第 4 章中所见，标准栈的大小允许 1000 到 3000 个递归步骤。由于高度为 1000 的完全平衡二叉树包含了 2^{1000} 个元素，所以你永远不会有足够的主内存来处理这样的一棵树。

这是一个好消息。但坏消息是，并非所有的二叉树都是完全平衡的。因为完全不平衡的二叉树实际上是一个单链表，它具有与列表相同的性能和相同的递归问题。这意味着要从树中取得最大的成效，你需要找到一种平衡它们的办法。

10.1.5 插入顺序

树的结构（即其平衡程度）取决于其元素的插入顺序。插入的方式与检索相同：

- 1 比较根的值与要插入的元素的值。如果它们相等，那你就完成了操作。由于插入的元素的值只能小于或大于根的值，因此相等就无法插入。但是请注意，现实情况有时会有所不同。如果待插入对象从树的顺序上看也许相等，但从其他标准上看并不相等，则可能需要把根的值替换为待插入的元素的值。很快你就会看到，这将是最常见的情况。

- 2 如果待插入的元素小于根，将其递归插入左分支中。
- 3 如果待插入的元素大于根，将其递归插入右分支中。

这个过程导致了一个非常有趣的现象：树的平衡取决于插入元素的顺序。很明显，插入有序元素会产生完全不平衡的树。而另一方面，许多插入顺序会生成相同的树。图 10.5 展示了产生同一棵树的可能插入顺序。

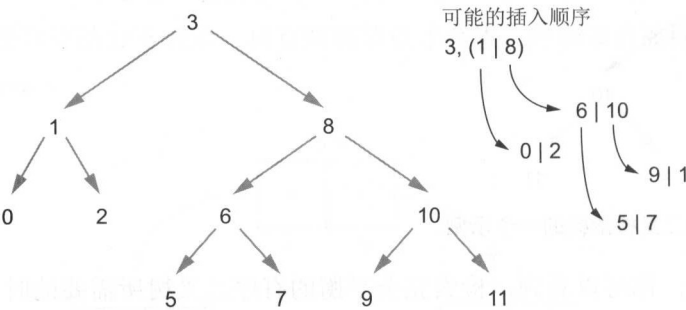


图 10.5 许多不同的插入顺序会生成相同的树。

10 个元素的集可以以 3 628 800 种不同的顺序插入一棵树中，但只会生成 16 796 棵不同的树。这些树会从完全平衡一直到完全不平衡。从更实际的角度上看，有序树对于存储和获取随机数据而言非常高效，但是对于存储和检索预先排好序的数据来说，它们非常糟糕。你很快就会学到如何解决这个问题。

10.1.6 树的遍历顺序

给定如图 10.5 所示的特定树，一个常见的用例就是遍历它，一个接一个地访问所有元素。在映射或折叠树时一般都是这样，而在树中搜索某个特定值时较少。当我们学习列表时，你学到了两种遍历的办法：从左到右或从右到左。树提供了更多的方式，我们将用递归和非递归来区分它们。

递归遍历顺序

思考图 10.5 中树的左分支。该分支本身是由根 1，左分支 0 和右分支 2 组成的树。你可以用 6 种顺序来遍历该树：

- 1, 0, 2
- 1, 2, 0
- 0, 1, 2

- 2, 1, 0
- 0, 2, 1
- 2, 0, 1

可以看到其中有三个命令与另外三个命令是对称的。1, 0, 2 和 1, 2, 0 是对称的。从根开始, 然后从左到右或从右到左访问两个分支。对于 0, 1, 2 和 2, 1, 0 也是如此, 它们只有分支的顺序不同, 还有 0, 2, 1 和 2, 0, 1。你只用考虑从左到右的方向 (因为另一个方向完全一样, 就像在镜子中看到的那样), 所以你还有三个顺序, 它们以根的位置命名:

- 前序 (1 0 2 或 1 2 0)
- 中序 (0 1 2 或 2 1 0)
- 后序 (0 2 1 或 2 0 1)

这些术语是在一个操作中的运算符的位置之后提出的。为了更好地类比, 想象一下用加号 (+) 替换根 (1), 生成如下样式:

- 前缀 (+ 0 2 或 + 2 0)
- 中缀 (0 + 2 或 2 + 0)
- 后缀 (0 2 + 或 2 0 +)

应用递归于整棵树, 这些顺序导致在遍历树时优先考虑高度, 通往如图 10.6 所示的遍历路径。请注意, 这种类型的遍历通常称为深度优先, 而不是逻辑上的高度优先。当讲整棵树时, 高度和深度是指根的高度和叶的深度。这两个值相等。

非递归遍历顺序

遍历树的另一种方法是首先访问一个完整的层级, 然后再进入下一个层级。还是可以从左到右或从右到左来完成。这种遍历称为层序遍历 (level-order traversal), 或广度优先搜索 (breadth-first search), 图 10.7 是一个示例。

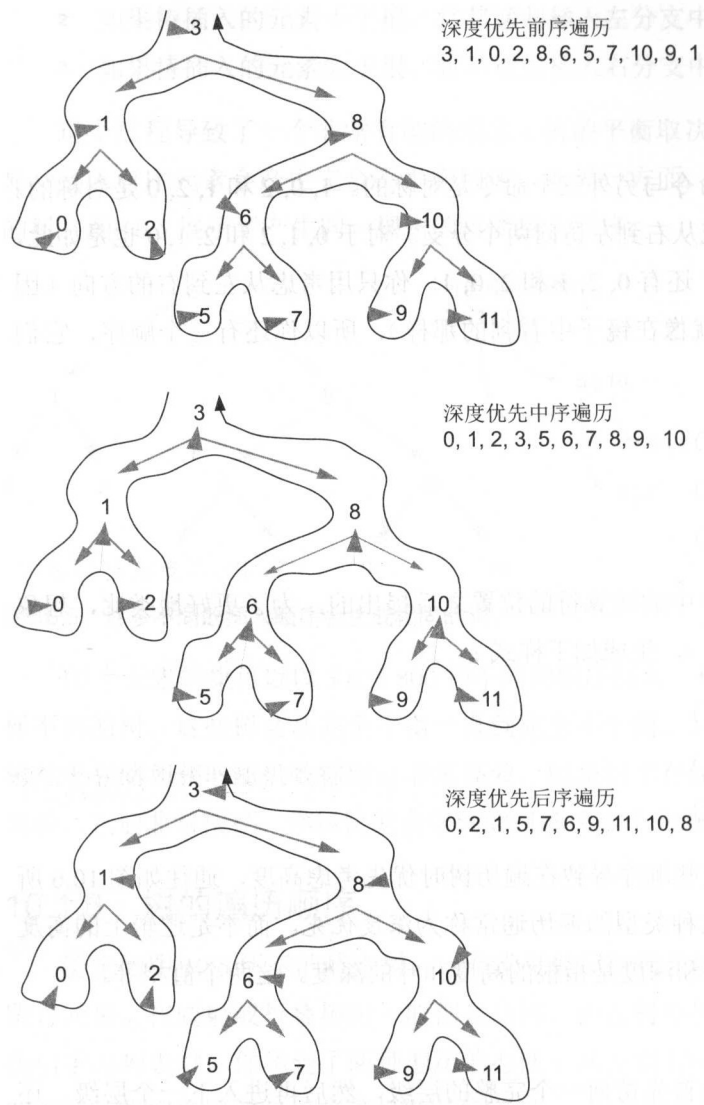


图 10.6 深度优先遍历包括在遍历树时优先考虑高度。可以应用三种主要顺序。

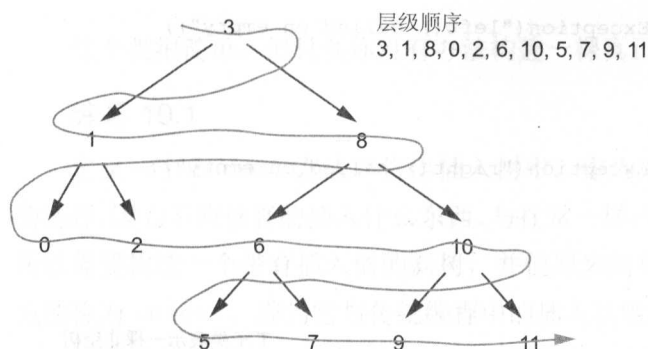


图 10.7 层级顺序包括在进入下一层级之前，先访问给定层级的所有元素。

10.2 实现二叉搜索树

我们将在本书中思考传统的二叉树，而不是叶树。二叉树的实现方式与单链表相同，有一个 head（称为 value）和两个 tail（分支，称为 left 和 right）。你将定义一个 Tree 抽象类以及名为 T 和 Empty 的两个子类。T 表示非空树，而 Empty 表示空树。清单 10.1 展示了最简的 Tree 实现。

清单 10.1 Tree 的实现

Tree 类被参数化，参数类型必须继承 Comparable

空树由非参数化的单例表示

公有的 value 方法返回根的值

left 和 right 方法都是私有的。它们只能在继承的内部类中使用

Empty 子类表示一棵空树

```
public abstract class Tree<A extends Comparable<A>> {  
  
    @SuppressWarnings("rawtypes")  
    private static Tree EMPTY = new Empty();  
    public abstract A value();  
    abstract Tree<A> left();  
    abstract Tree<A> right();  
  
    private static class Empty<A extends Comparable<A>> extends Tree<A> {  
  
        @Override  
        public A value() {  
            throw new IllegalStateException("value() called on empty");  
        }  
  
        @Override  
        Tree<A> left() {
```

```

        throw new IllegalStateException("left() called on empty");
    }

    @Override
    Tree<A> right() {
        throw new IllegalStateException("right() called on empty");
    }

    @Override
    public String toString() {
        return "E";
    }
}

```

T 子类表示一棵非空树

```
private static class T<A extends Comparable<A>> extends Tree<A> {
```

```

    private final Tree<A> left;
    private final Tree<A> right;
    private final A value;

```

```

    private T(Tree<A> left, A value, Tree<A> right) {
        this.left = left;
        this.right = right;
        this.value = value;
    }

```

```

    @Override
    public A value() {
        return value;
    }

```

```

    @Override
    Tree<A> left() {
        return left;
    }

```

```

    @Override
    Tree<A> right() {
        return right;
    }

```

```

    @Override
    public String toString() {
        return String.format("(T %s %s %s)", left, value, right);
    }
}

```

```

@SuppressWarnings("unchecked")
public static <A extends Comparable<A>> Tree<A> empty() {
    return EMPTY;
}
}

```

empty 方法返回 EMPTY 单例，
为了防止关于未检查类型分配的
编译器警告从 Tree 类中泄
露出去，加上了定义

这个类很简单，但只要你没有办法构建一棵真正的树，它便没什么用。

练习 10.1

定义一个 `insert` 方法以将值插入树中。作为方法名称的 `insert` 并不是很好的选择，因为不应该真的插入什么东西。与往常一样，`Tree` 结构是不可变和持久化的，所以需要构造一个带有插入值的新树，并使原来的树保持不变。但是通常还是将该方法称为 `insert`，因为它与传统编程中的插入功能相同。

如果该值等于根，则你必须返回一棵新树，以插入的值为根并保持原来的两个分支不变。否则，在左分支中插入小于根的值，在右分支中插入大于根的值。在 `Tree` 父类中声明该方法，并在两个子类中实现。方法签名如下：

```
public abstract Tree<A> insert(A a);
```

答案 10.1

`Empty` 的实现构造一个新的 `T`，以插入的值为根并以两棵空树为分支：

```
public Tree<A> insert(A insertedValue) {
    return new T<>(empty(), insertedValue, empty());
}
```

`T` 的实现有点复杂。首先，它比较插入值与根。如果插入值较小，则用当前根与当前的右分支构造新的 `T`。左分支是将值递归插入原来的左分支中的结果。

如果值大于根，则用当前根与当前的左分支构造新的 `T`。右分支是将值递归插入原来的右分支中的结果。

最后，如果该值等于根，则返回一棵新树，由作为根的插入值和两个原来的不变分支所组成：

```
public Tree<A> insert(A insertedValue) {
    return insertedValue.compareTo(this.value) < 0
        ? new T<>(left.insert(insertedValue), this.value, right)
        : insertedValue.compareTo(this.value) > 0
            ? new T<>(left, this.value, right.insert(insertedValue))
            : new T<>(this.left, insertedValue, this.right);
}
```

请注意，这与 `Java` 的 `TreeSet` 有所不同，如果你尝试插入的元素等于集合中已存在的元素，则 `TreeSet` 不会改变。尽管这种行为对于可变元素而言或许可以接受，但是当元素不可变时，这是不可接受的。你可能会认为，使用相同的左分支，

相同的右分支和与当前根相同的根来构建新的 T 实例，只不过是浪费时间和内存空间，因为你可以简单地返回 this。返回 this 等价于返回

```
new T<>(this.left, this.value, this.right)
```

如果这就是你想要的，返回 this 会是一个很好的优化。这样做可以工作，但是获得与改变树元素相同的结果实在无趣。在插入某些属性已更改的相等元素之前，你不得不先删除该元素。你会在第 11 章中实现 map 时碰到这种用例。

你可能想知道是否应该实现栈安全的递归，因为 insert 方法是递归的。如前所述，使用平衡树的话就没有必要，因为高度（决定了最大递归步数）通常远低于树的大小。但你也看到了并不总是如此，尤其是当待插入的元素已经排好序时。这可能最终会导致一棵只有一个分支的树，其高度等于其大小（减 1），并且会栈溢出。

不过现在，你不用为这个问题而烦恼。不必实现栈安全的递归操作，你将会找到一种自动平衡树的办法。你正着手的简单树只是为学习而生，它永远也不会用于生产。但平衡树实现起来比较复杂，所以 you 将从简单的不平衡树开始。

练习 10.2

一个经常用于树的操作是检查树中是否存在特定元素。实现一个执行此检查的 member 方法。其签名如下：

```
boolean member(A a)
```

提示

将其实现为 Tree 父类的抽象方法和每个子类的特定实现。

答案 10.2

我们先从 T 子类的实现开始。你需要比较参数与树的值（指的是根的值）。如果参数较小，则递归地比较左分支。如果参数较大，则递归地比较右分支。如果值与参数相等，只需返回 true 即可：

```
public boolean member(A value) {
    return value.compareTo(this.value) < 0
        ? left.member(value)
        : value.compareTo(this.value) > 0
        ? right.member(value)
        : true;
}
```

请注意，这段代码可以进行如下简化：

```
public boolean member(A value) {
    return value.compareTo(this.value) < 0
        ? left.member(value)
        : value.compareTo(this.value) == 0 || right.member(value);
}
```

但是你可能会觉得第一个版本更整洁。Empty 的实现当然是返回 false。

练习 10.3

为了简化树的创建，请编写一个静态方法，它接收一个变长参数，并将所有元素插入一棵空树中。下面是它的签名：

```
public static <A extends Comparable<A>> Tree<A> tree(A... as)
```

提示

首先实现一个以列表为参数的方法，然后根据 list 方法定义变长参数方法。

答案 10.3

这更像是关于列表而非关于树的练习。解决方案如下：

```
public static <A extends Comparable<A>> Tree<A> tree(List<A> list) {
    return list.foldLeft(empty(), t -> t::insert);
}

@SafeVarargs
public static <A extends Comparable<A>> Tree<A> tree(A... as) {
    return tree(List.list(as));
}
```

练习 10.4

编写计算树的 size 和 height 的方法。这是它们在 Tree 类中的签名：

```
public abstract int size();
public abstract int height();
```

答案 10.4

当然，size 在 Empty 中的实现返回 0。正如我先前所说，height 方法在 Empty 中的实现返回 -1。size 方法在 T 类中的实现返回 1 加上每个分支的大小。height 方法的实现返回 1 加上两个分支的最大 height：


```
public int size() {  
    return 1 + left.size() + right.size();  
}  
  
public int height() {  
    return 1 + Math.max(left.height(), right.height());  
}
```

由此可见为什么一棵空树的高度必须等于 -1。如果它为 0，则高度将等于路径中的元素数量，而不是线段数量。

请注意，这些方法仅用于说明。实际上，你可以像 List 中的 length 一样记录高度和大小。查看本书附带的代码可以提醒你如何做到这一点。

练习 10.5

编写 max 和 min 方法来计算树中包含的最大值和最小值。

提示

思考在 Empty 类中的方法应该返回什么。

答案 10.5

当然，空树没有最小值或最大值。答案是返回一个 Result<A>，而 Empty 的实现将返回 Result.empty()。T 类的实现有点棘手。对于 max 方法，解决方案是返回右分支的最大值。如果右分支不为空，这就是一个递归调用。如果右分支为空，你将得到 Result.Empty。于是你知道 max 值就是当前树的值，因此你可以直接在 right.max() 方法的返回值上调用 orElse 方法：

```
public Result<A> max() {  
    return right.max().orElse(() -> Result.success(value));  
}
```

回想一下，orElse 方法对其参数惰性求值，意味着它接收一个 Supplier<Result<A>>。当然，min 方法是完全对称的：

```
public Result<A> min() {  
    return left.min().orElse(() -> Result.success(value));  
}
```

10.3 从树中删除元素

与单链接列表不同，树允许你像在练习 10.2 中开发 `member` 方法时看到的那样来检索指定的元素。这样也可以从树中删除一个指定的元素。

练习 10.6

编写一个从树中删除元素的 `remove` 方法。该方法以一个元素为参数。如果树中存在该元素则会将其删除，并返回一棵没有该元素的新树。当然，这棵新树将遵守左分支上的所有元素都小于根并且右分支上的所有元素都大于根的要求。如果该元素不在树中，则方法将返回未变化的树。方法签名如下

```
Tree<A> remove(A a)
```

提示

你需要定义一个方法以合并两棵特别的树，其中一棵的所有元素都大于或小于另一棵的所有元素。你还需要一个 `isEmpty` 方法，在 `Empty` 类中返回 `true`，在 `T` 类中返回 `false`。

答案 10.6

`Empty` 的实现当然只是返回 `this`，什么都删除不了。对于 `T` 子类的实现而言，你需要实现的算法如下：

- 如果 `a < this`，从左边删除。
- 如果 `a > this`，从右边删除。
- 否则，删除根。合并左右分支，舍弃掉根，并返回结果。

这是一个简化版的合并，因为你知道左分支的所有元素都小于右分支的所有元素。

首先你必须定义 `merge` 方法。在 `Tree` 类中定义一个抽象方法：

```
protected abstract Tree<A> removeMerge(Tree<A> ta)
```

`Empty` 类中的实现简单地返回未变化的参数，因为合并 `ta` 与一棵空树的结果还是 `ta`：

```
protected Tree<A> removeMerge(Tree<A> ta) {
```

```
return ta;
}
```

T 的实现使用以下算法：

- 如果 ta 为空，则返回 this (this 不会为空)。
- 如果 ta < this，则在左分支中合并 ta。
- 如果 ta > this，则在右分支中合并 ta。

实现如下：

```
protected Tree<A> removeMerge(Tree<A> ta) {
    if (ta.isEmpty()) {
        return this;
    }
    if (ta.value().compareTo(value) < 0) {
        return new T<>(left.removeMerge(ta), value, right);
    } else if (ta.value().compareTo(value) > 0) {
        return new T<>(left, value, right.removeMerge(ta));
    }
    throw new IllegalStateException("We shouldn't be here");
}
```

请注意，如果两棵树的根相等，该方法将抛出异常，但不应该发生这种事情，因为要合并的两棵树是原来同一棵树的左右分支。

现在你可以编写 remove 方法了：

```
public Tree<A> remove(A a) {
    if (a.compareTo(this.value) < 0) {
        return new T<>(left.remove(a), value, right);
    } else if (a.compareTo(this.value) > 0) {
        return new T<>(left, value, right.remove(a));
    } else {
        return left.removeMerge(right);
    }
}
```

10.4 合并任意树

在 10.3 节中，你使用的合并方法能够合并的树是有限制的，即其中一棵树中的所有值都小于另一棵树的所有值。合并树等价于连接列表。你需要一个更通用的方法合并任意的树。

练习 10.7 (难)

到目前为止, 你只合并了一棵树中所有元素都大于另一棵树中所有元素的树。编写一个合并任意树的 merge 方法, 其签名如下:

```
public abstract Tree<A> merge(Tree<A> a);
```

答案 10.7

Empty 的实现只是返回它的参数:

```
public Tree<A> merge(Tree<A> a) {  
    return a;  
}
```

T 子类的实现将会使用以下算法, 其中的 this 表示定义了方法的树:

- 如果参数树为空, 则返回 this。
- 如果参数的根大于 this 的根, 则删除参数树的左分支, 并将该结果与 this 的右分支合并。然后将结果与参数的左分支合并。
- 如果参数的根小于 this 的根, 则删除参数树的右分支, 并将该结果与 this 的左分支合并。然后将结果与参数的右分支合并。
- 如果参数的根等于 this 的根, 则合并参数的左分支与 this 的左分支, 再合并参数的右分支与 this 的右分支。

下面是算法的实现:

```
public Tree<A> merge(Tree<A> a) {  
    if (a.isEmpty()) {  
        return this;  
    }  
    if (a.value().compareTo(this.value()) > 0) {  
        return new T<>(left, value, right.merge(new T<>(empty(),  
            a.value(), a.right()))).merge(a.left());  
    }  
    if (a.value().compareTo(this.value()) < 0) {  
        return new T<>(left.merge(new T<>(a.left(), a.value(),  
            empty())), value, right).merge(a.right());  
    }  
    return new T<>(left.merge(a.left()), value, right.merge(a.right()));  
}
```

该算法如图 10.8 至图 10.17 所示。

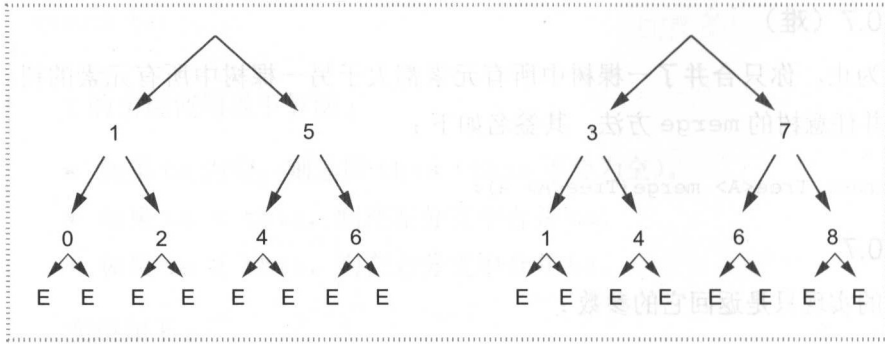


图 10.8 待合并的两棵树。左边是 this 树，右边是参数树。

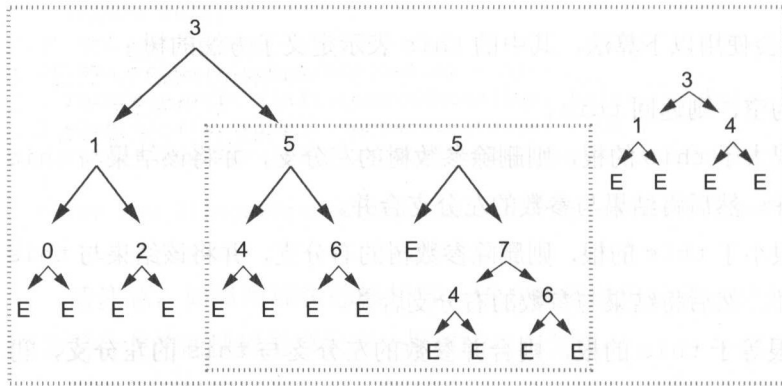


图 10.9 参数树的根大于 this 树的根。合并 this 树的右分支与删除了左分支的参数树。(合并操作由虚线框表示。)

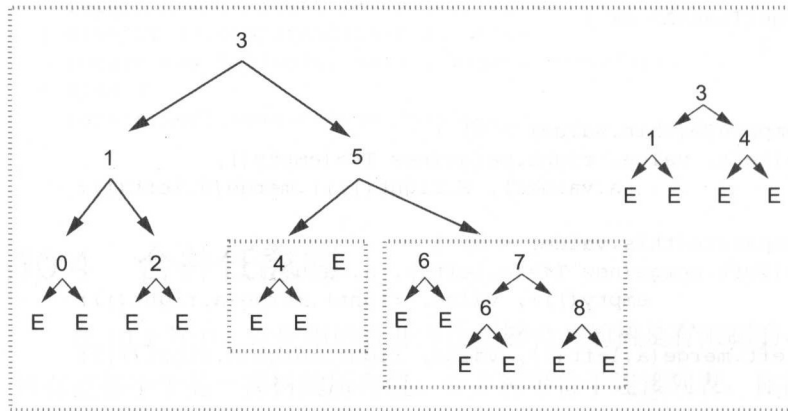


图 10.10 待合并的每棵树的根都相等，你可以用这个值作为合并的结果。左分支将是合并两个左分支的结果，右分支将是合并两个右分支的结果。

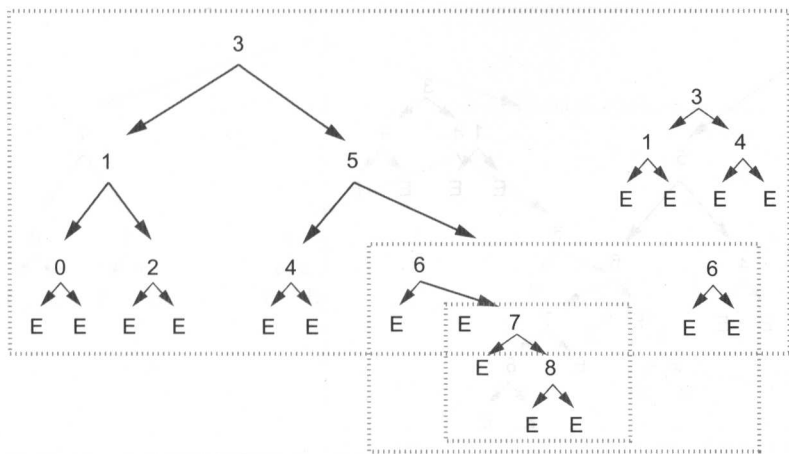


图 10.11 对于左分支而言,合并空树就是小事一桩,只需返回原来的树(根 4 和两个空分支)即可。对于右分支而言,第一棵树有空分支和作为根的 6,第二棵树有作为根的 7,因此删除根为 7 的树的左分支,并用结果合并根为 6 的树的空的右分支。删除的左分支将合并上一次合并的结果。请注意,右侧以 6 为根的树来自以 7 为根的树,它已经被替换为一棵空树。

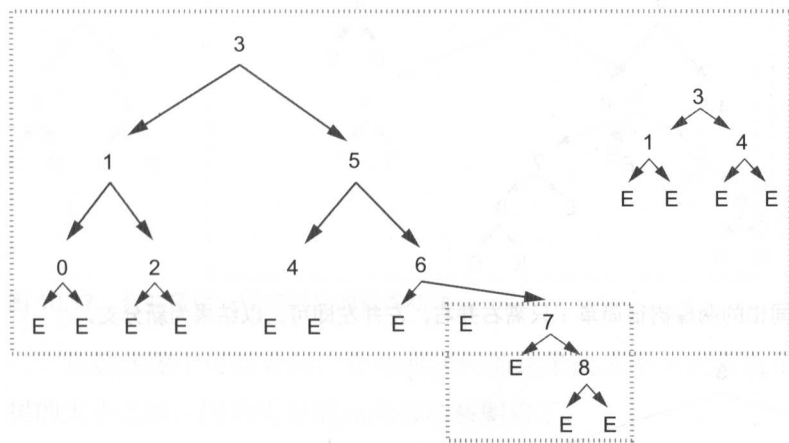


图 10.12 待合并的两棵树的根相等(6),所以要合并分支(左和左,右和右)。因为要合并的树的两个分支都是空的,实际上不需要做什么。

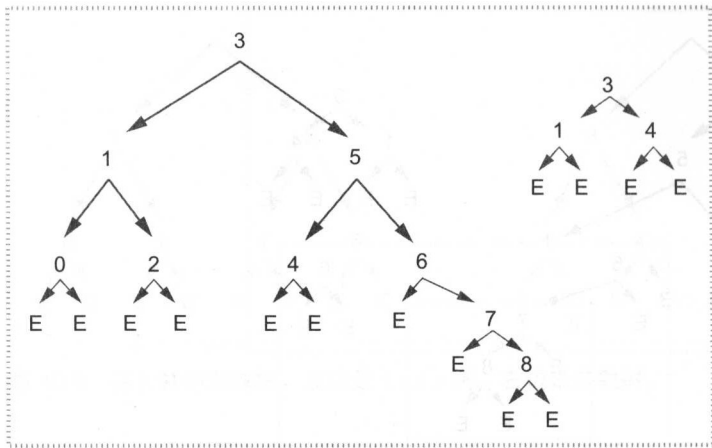


图 10.13 合并一棵空树的简单结果就是待合并的树。你就剩余两棵待合并的树，它们的根相同。

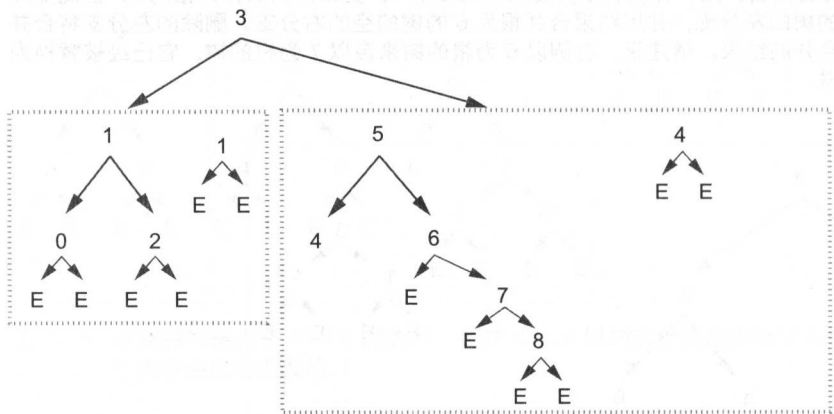


图 10.14 合并拥有相同根的两棵树很简单：只需右并右，左并左即可，以结果为新分支。

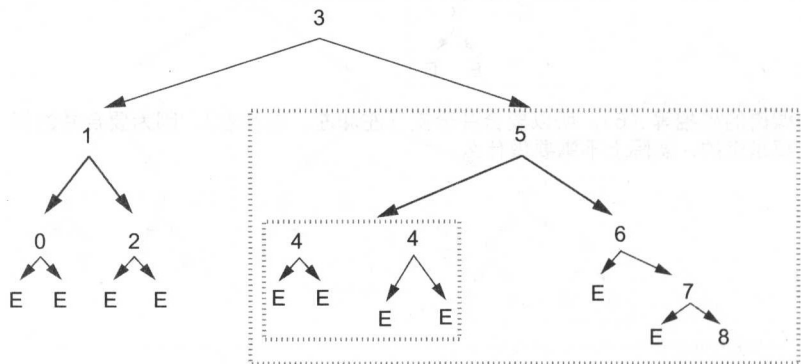


图 10.15 合并左侧轻而易举，因为根是相等的，并且待合并的树的两个分支都是空的。在右侧，待合并的树具有较小的根（4），因此删除右分支（E）合并原来的树的左分支的剩余部分。

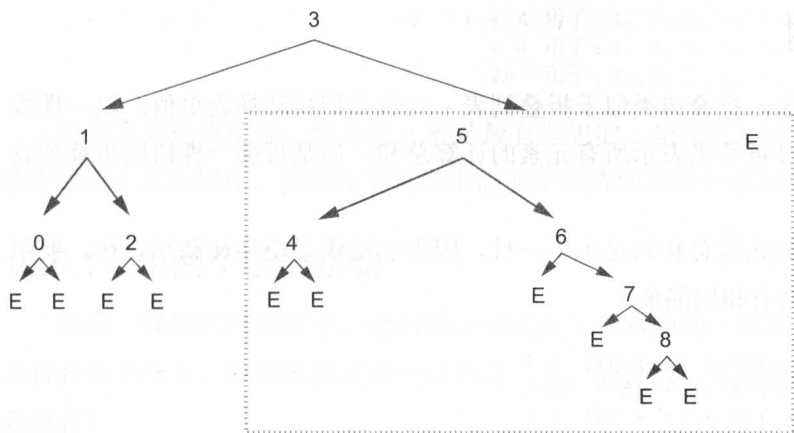


图 10.16 合并两棵相同的树，不需解释。

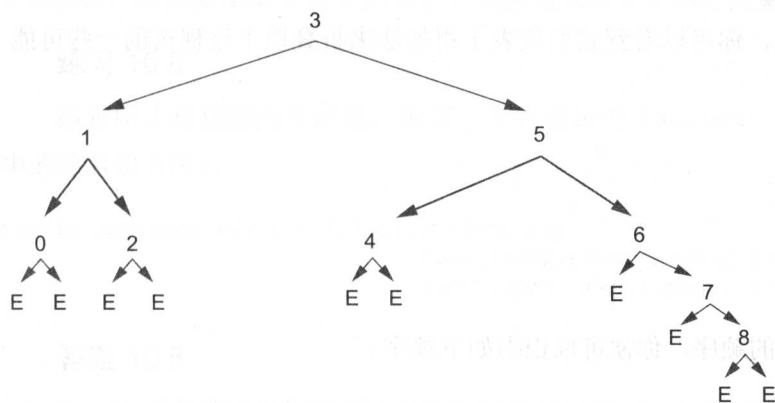


图 10.17 合并最后一棵空树后的最终结果。

从这些图中可以看到，合并两棵树的結果的大小（元素数量）可以小于原来的树的大小之和，因为重复的元素自动被删除了。

另一方面，结果的高度高于你的预期。合并两棵高度为 3 的树可能会导致结果为高度为 5 的树。很容易看出，最佳高度不应高于 $\log_2(\text{size})$ 。换句话说，最佳高度是 2 的乘幂中大于结果大小的最小值。在这个例子中，原来的两棵树大小为 7，高度为 3。合并后的树大小为 9，最佳高度应该为 4 而不是 5。在这样的小例子中，可能不是什么问题。但是，当你合并大树时，可能会得到极不平衡的树，从而导致性能不够理想，甚至可能在使用递归方法时栈溢出。

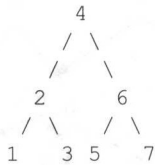
10.5 折叠树

本节与折纸无关。折叠树类似于折叠列表；它包括把树转换为单值。以一棵数字树为例，可以通过折叠来表示所有元素的计算总和。但是折叠一棵树比折叠列表要复杂得多。

计算整型树中的元素总和只是小事一桩，因为加法满足交换律和结合律。换句话说，以下表达式具有相同的值：

* $((1 + 3) + 2) + ((5 + 7) + 6)) + 4$
* $4 + ((2 + (1 + 3)) + (6 + (5 + 7)))$
* $((7 + 5) + 6) + ((3 + 1) + 2)) + 4$
* $4 + ((6 + (7 + 5)) + (2 + (3 + 1)))$
* $(1 + (2 + 3)) + (4 + (5 + (6 + (7))))$
* $(7 + (6 + 5)) + (4 + (3 + (2 + 1)))$

观察这些表达式，你可以看到它们代表了用加法来折叠以下这棵树的一些可能结果：



只考虑处理元素的顺序，你便可以识别如下顺序：

- 左后序
- 左前序
- 右后序
- 右前序
- 左中序
- 右中序

请注意，左和右的意思是从左边开始还是从右边开始。你可以通过计算每个表达式的结果来验证这一点。例如，第一个表达式可以如下化简：

$((1 + 3) + 2) + ((5 + 7) + 6)) + 4$
 $((\quad 4 \quad + 2) + ((5 + 7) + 6)) + 4$ 用了：1、3
 $(\quad 6 \quad + ((5 + 7) + 6)) + 4$ 用了：1、3、2
 $(\quad 6 \quad + (\quad 12 \quad + 6)) + 4$ 用了：1、3、2、5、7

(6 + 18) + 4 用了: 1、3、2、5、7、6
 24 + 4 用了: 1、3、2、5、7、6
 28 用了: 1、3、2、5、7、6、4

还有其他的可能性,但是这6种是最有意思的。虽然它们在加法上等价,但可能不适用于其他操作,例如向字符串中添加字符或向列表中添加元素。

10.5.1 用两个函数折叠

折叠一棵树的问题在于,递归的方式实际上是双向的。你可以使用给定的操作来折叠每个分支,但是需要把两个结果合二为一的办法。这是否提醒了你并行化列表折叠?

是的,你需要一个额外的操作。如果折叠 $\text{Tree}\langle A \rangle$ 所需的操作是从 B 到 A 到 B 的函数,则你需要从 B 到 B 到 B 的附加函数来合并左右的结果。

练习 10.8

给定刚才描述的两个函数,编写一个折叠树的 `foldLeft` 方法。它在 `Tree` 类中的签名如下所示:

```
public abstract <B> B foldLeft(B identity,
                                Function<B, Function<A, B>> f,
                                Function<B, Function<B, B>> g)
```

答案 10.8

`Empty` 子类的实现很直观,只需返回单位元即可。`T` 子类的实现稍微困难一些。你需要递归计算每个分支的折叠,然后将结果与根组合起来。问题是每个分支的折叠返回一个 B , 可根是一个 A , 而你却没有从 A 到 B 的函数。可能的解决方案如下:

- 1 给定两个 B 值,递归地折叠左分支和右分支。
- 2 用 g 函数组合这两个 B 值,然后将此结果与根组合起来并返回新结果。

这是一个方案:

```
public <B> B foldLeft(B identity,
                      Function<B, Function<A, B>> f,
                      Function<B, Function<B, B>> g) {
    return g.apply(right.foldLeft(identity, f, g))
        .apply(f.apply(left.foldLeft(identity, f, g)).apply(this.value));
}
```

简单吗？并非如此。问题是函数 g 是从 B 到 B 到 B 的函数，所以你不小心就会把参数搞反了：

```
public <B> B foldLeft(B identity,
    Function<B, Function<A, B>> f,
    Function<B, Function<B, B>> g) {
    return g.apply(*left*.foldLeft(identity, f, g))
        .apply(f.apply(*right*.foldLeft(identity, f, g)).apply(this.value));
}
```

这是一个问题吗？是的。如果你用符合交换律的操作来折叠列表，如加法，则结果将不会更改。但是，如果用不符合交换律的操作，那你就遇上了麻烦。最终的结果是，两种方案会给你不同的结果。以如下函数为例，

```
Tree.tree(4, 2, 6, 1, 3, 5, 7)
    .foldLeft(List.list(), list -> a -> list.cons(a),
        x -> y -> y.concat(x)).toString();
```

使用第一种方案将生成以下结果，

```
[4, 2, 1, 3, 6, 5, 7, NIL]
```

而第二种方案的结果如下：

```
[4, 6, 7, 5, 2, 3, 1, NIL]
```

哪个才是正确的结果？你可以通过交换第二个函数的参数来找到原来的结果：

```
Tree.tree(4, 2, 6, 1, 3, 5, 7)
    .foldLeft(List.list(), list -> a -> list.cons(a),
        x -> y -> x.concat(y)).toString();
```

其实这两个列表虽然顺序不同，但都表示相同的树。图 10.18 展示了这两种情况。

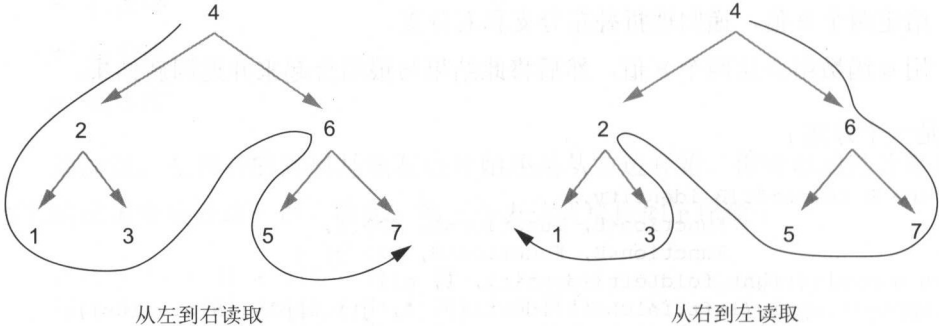


图 10.18 从左到右和从右到左读取树。

在本书附带的代码中可以找到这两个例子。请注意，这不同于 List 类的 foldLeft 和 foldRight。从右到左的折叠实际上是对反转列表的左折叠。右折叠看起来就像这样：

```
@Override
public <B> B foldRight(B identity,
    Function<A, Function<B, B>> f,
    Function<B, Function<B, B>> g) {
    return g.apply(f.apply(this.value).apply(left.foldRight(identity, f, g)))
        .apply(right.foldRight(identity, f, g));
}
```

因为有很多遍历顺序，所以许多可能的实现将会对不符合交换律的操作给出不同的结果。你将在本书附带的代码注释中找到示例。

10.5.2 用一个函数折叠

也可以使用接收一个附加参数的函数来折叠，即如同从 B 到 A 到 B 到 B 的函数。再强调一遍，取决于遍历的顺序，将会有许多种可能的实现。

练习 10.9

编写三个折叠树的方法：foldInOrder、foldPreOrder 和 foldPostOrder。对应于图 10.18 中的树，元素应该如下处理。

- 中序：1 2 3 4 5 6 7
- 前序：4 2 1 3 6 5 7
- 后序：1 3 2 5 7 6 4

方法签名如下：

```
<B> B foldInOrder(B identity, Function<B, Function<A, Function<B, B>>> f);
<B> B foldPreOrder(B identity, Function<A, Function<B, Function<B, B>>> f);
<B> B foldPostOrder(B identity, Function<B, Function<B, Function<A, B>>> f);
```

答案 10.9

答案如下。Empty 的实现都返回单位元。T 类的实现如下：

```
public <B> B foldInOrder(B identity,
    Function<A, Function<B, B>>> f) {
    return f.apply(left.foldInOrder(identity, f))
        .apply(value).apply(right.foldInOrder(identity, f));
}
```

```

}

public <B> B foldPreOrder(B identity,
                        Function<A, Function<B, Function<B, B>>> f) {
    return f.apply(value).apply(left.foldPreOrder(identity, f))
        .apply(right.foldPreOrder(identity, f));
}

public <B> B foldPostOrder(B identity,
                        Function<B, Function<B, Function<A, B>>> f) {
    return f.apply(left.foldPostOrder(identity, f))
        .apply(right.foldPostOrder(identity, f)).apply(value);
}

```

10.5.3 选择哪种折叠的实现

你现在编写了 5 种不同的折叠方法。应该选用哪一种？为了回答这个问题，让我们来思考一个折叠方法应该具有的属性。

折叠数据结构的方式与其构建方式之间有一种联系。你可以通过始于一个空元素并逐个添加元素来构建数据结构。这与折叠相反。理想情况下，应该能够使用特定参数来折叠一个结构，使你可以将折叠转换为一个恒等函数。对于列表而言，如下所示：

```
list.foldRight(List.list(), i -> l -> l.cons(i));
```

你也可以使用 `foldLeft`，但是这个函数会稍微复杂一些：

```
list1.foldLeft(List.list(), l -> i -> l.reverse().cons(i).reverse());
```

（这并不奇怪；如果查看 `foldRight` 的实现，你会看到它的内部使用了 `foldLeft` 和 `reverse`。）

可以同样折叠树吗？为了实现这一点，你需要一种通过组合一棵左树、一个根和一棵右树来构建树的新方式。这样，你就可以在三种折叠方法中任选其一，让它只接收一个函数参数。

练习 10.10（难）

创建一个通过组合两棵树和一个根来创建一棵新树的方法。其签名如下：

```
Tree<A> tree(Tree<A> left, A a, Tree<A> right)
```

这个方法应该允许你在以下三种折叠方法中任选其一来重新构造与原来的树相

同的树：foldPreOrder、foldInOrder 和 foldPostOrder。

提示

你需要分别处理两种情况。如果要合并的树已排序，即如果第一棵树的最大值小于根，并且第二棵树的最小值大于根，那你就可以使用 T 的构造函数简单地组合它们三个。否则，你应该转到另一种构建结果的方式中。

答案 10.10

实现这个方法有几种方式。其一是首先定义一个方法来检查两棵树是否已排序。为此，你可以先定义方法来返回值的比较结果：

```
public static <A extends Comparable<A>> boolean lt(A first, A second) {
    return first.compareTo(second) < 0;
}

public static <A extends Comparable<A>> boolean lt(A first, A second,
                                                    A third) {
    return lt(first, second) && lt(second, third);
}
```

接下来可以定义实现了树的比较的 ordered 方法：

```
public static <A extends Comparable<A>> boolean ordered(Tree<A> left,
                                                         A a, Tree<A> right) {
    return left.max().flatMap(lMax -> right.min().map(rMin ->
        lt(lMax, a, rMin))).getOrElse(left.isEmpty() && right.isEmpty())
    || left.min().flatMapEmpty().flatMap(ignore -> right.min().map(rMin ->
        lt(a, rMin))).getOrElse(false)
    || right.min().flatMapEmpty().flatMap(ignore -> left.max().map(lMax ->
        lt(lMax, a))).getOrElse(false);
}
```

如果两棵树都不为空，并且左树的 max、a 和右树的 min 都已排序，则第一个检查（第一个 || 运算符之前）返回 true。第二个检查和第三个检查处理左树或右树为空（但两棵不都为空）的情况。请注意，如果 Result 为空，则 Result.mapEmpty 方法返回 Success<Nothing>，否则返回一个失败。

使用这个方法来编写 tree 方法就很简单了：

```
public static <A extends Comparable<A>> Tree<A> tree(Tree<A> t1,
                                                       A a, Tree<A> t2) {
    return ordered(t1, a, t2)
        ? new T<>(t1, a, t2)
        : ordered(t2, a, t1)
}
```

```

? new T<>(t2, a, t1)
: Tree.<A>empty().insert(a).merge(t1).merge(t2);
}

```

请注意，如果树未排序，则在转到常规的插入 / 合并算法之前，先检查倒序。

现在，你可以折叠一棵树并获得与原来的树相同的树（只要使用了正确的函数）。

你会在本书附带的测试代码中找到以下示例：

```

tree.foldInOrder(Tree.<Integer>empty(),
                  t1 -> i -> t2 -> Tree.tree(t1, i, t2));
tree.foldPostOrder(Tree.<Integer>empty(),
                   t1 -> t2 -> i -> Tree.tree(t1, i, t2));
tree.foldPreOrder(Tree.<Integer>empty(),
                  i -> t1 -> t2 -> Tree.tree(t1, i, t2));

```

你还可以定义只接收一个双参函数的折叠方法，就像你在 List 中所做。诀窍是首先将树转换为一个列表，如 foldLeft 的例子所示：

```

public <B> B foldLeft(B identity, Function<B, Function<A, B>> f) {
    return toListPreOrderLeft().foldLeft(identity, f);
}

protected List<A> toListPreOrderLeft() {
    return left().toListPreOrderLeft()
        .concat(right().toListPreOrderLeft()).cons(value);
}

```

这也许不是最快的实现，但它仍可能有用。

10.6 映射树

可以像列表一样映射树，但是映射树会更复杂一些。也许将函数应用于树的每个元素看起来微不足道，但并非如此。问题在于不是所有的函数都会保留顺序。给整型树中的所有元素都加上一个给定值没什么问题，但要是树可能包含负值，则会使函数 $f(x) = x * x$ 变得更加复杂，因为简单地“原地”应用函数的结果不会是二叉搜索树。

练习 10.11

定义一个树的 map 方法。尽量保留树的结构。例如，通过对值求平方来映射整型树可能会生成具有不同结构的树，但是通过加上常量来映射就不会。

答案 10.11

使用任一折叠方法都是非常直观的。使用不同的折叠方法会有几种可能的实现。一个示例如下：

```
public <B extends Comparable<B>> Tree<B> map(Function<A, B> f) {  
    return foldInOrder(Tree.<B>empty(),  
        t1 -> i -> t2 -> Tree.tree(t1, f.apply(i), t2));  
}
```

当然，Empty 的实现返回 empty()（不是 this，因为类型无效）。

10.7 平衡树

正如我先前所说，树会在平衡过以后很好地工作，这意味着从根到叶元素的所有路径都具有相同的长度。在一棵完全平衡的树中，长度相差不会超过 1，只有在较深的层级没有被填满时才会不同。（只有大小为 $2n + 1$ 的完全平衡树从根到叶元素的所有路径才相同。）

使用不平衡的树可能导致性能不佳，因为操作所需时间与树的大小而不是 $\log_2(\text{size})$ 成正比。更具戏剧性的是，不平衡的树可能会在使用递归操作时引起栈溢出。有两种办法来避免这个问题：

- 使不平衡的树平衡。
- 使用自平衡树。

一旦你有办法来平衡树，很容易就可以通过在每次可能改变树结构的操作之后自动启动平衡过程。

10.7.1 旋转树

在平衡树之前，你需要知道如何逐步改变树的结构。所使用的技术称为旋转树，如图 10.19 和图 10.20 所示。

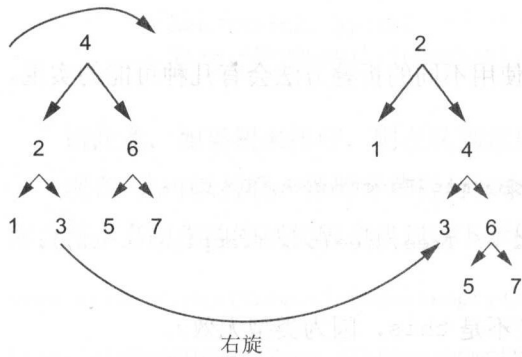


图 10.19 向右旋转树。在旋转中，2 和 3 之间的线被替换为 2 和 4 之间的线，因此元素 3 被移动并成为 4 的左元素。

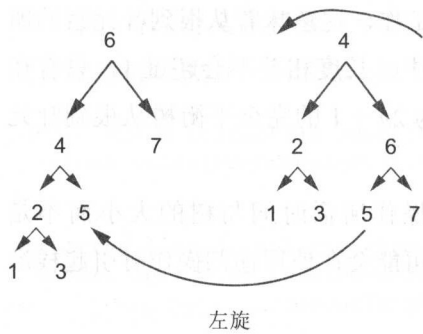


图 10.20 向左旋转树。6 的左元素变成 4(以前是 6 的父元素)，因此 5 被移动并成为 4 的右元素。

练习 10.12

编写 `rotateRight` 和 `rotateLeft` 方法以在两个方向上旋转树。小心维持分支顺序。左元素必须总是小于根，而右元素必须总是大于根。在父类中声明抽象方法。由于这两个方法只能在 `Tree` 类中使用，将它们设置为 `protected`。以下是父类中的签名：

```
protected abstract Tree<A> rotateLeft();
protected abstract Tree<A> rotateRight();
```

答案 10.12

`Empty` 的实现只是简单地返回 `this`。在 `T` 类中，以下是右旋的步骤：

- 1 测试左分支是否为空。

- 2 如果左分支为空，返回 `this` 即可，因为右旋包含了将左元素提升为根。（你不能提升一棵空树。）
- 3 如果左元素不为空，那就让它成为根，所以以 `left.value` 为根来创建一个新的 `T`。左元素的左分支变为新树的左分支。对于右分支，你可以这样来构建一棵新树：以原来的根为根，原来的左侧的右分支为左分支，原来的右侧为右分支。

左旋是对称的：

```
protected Tree<A> rotateLeft() {
    return right.isEmpty()
        ? this
        : new T<>(new T<>(left, value, right.left()),
                  right.value(), right.right());
}

protected Tree<A> rotateRight() {
    return left.isEmpty()
        ? this
        : new T<>(left.left(), left.value(),
                  new T<>(left.right(), value, right));
}
```

要说清楚似乎很复杂，但其实相当简单。只需比较代码与图即可看到发生了什么情况。

如果尝试多次旋转树，你将到达一个分支为空的点，这时树就不能再以相同方向旋转了。

练习 10.13

为了平衡树，你还需要将树转换为有序列表的方法。编写一个方法将树改变为从右到左的已排序列表（就是降序）。如果你想尝试更多的练习，尽管去定义从左到右排序的方法，以及前排序和后排序的方法。

这是 `toListInOrderRight` 方法的签名：

```
public List<A> toListInOrderRight()
```

答案 10.13

这很简单，并且与列表而非树更加相关。`Empty` 的实现只是返回一个空列表而已。你可能会想到以下实现：

```
public List<A> toListInOrderRight() {
    return right.toListInOrderRight().concat(List.list(value))
        .concat((left.toListInOrderRight()));
}
```

不幸的是，如果树非常不平衡，这个方法会栈溢出。你需要用这个方法平衡一棵树，所以如果它不能用于不平衡的树，那就太不幸了！

这是一个栈安全的递归版本：

```
public List<A> toListInOrderRight() {
    return unBalanceRight(List.list(), this).eval();
}

private TailCall<List<A>> unBalanceRight(List<A> acc, Tree<A> tree) {
    return tree.isEmpty()
        ? TailCall.ret(acc)
        : tree.left().isEmpty()
            ? TailCall.sus(() ->
                unBalanceRight(acc.cons(tree.value()), tree.right()))
            : TailCall.sus(() ->
                > unBalanceRight(acc, tree.rotateRight()));
```

① 将树添加到累加器列表中

② 旋转树直到左分支为空

`unBalanceRight` 方法简单地右旋该树，直到左分支为空②。然后在将树的值添加到累加器列表①之后，递归地调用自身来同样处理所有的右子树。最终，树的参数为空，于是方法返回列表累加器。

10.7.2 使用 DSW 算法平衡树

DSW (Day-Stout-Warren) 算法很简单。首先，将树转换为完全不平衡的树。然后旋转直到树完全平衡。将树转换为一棵完全不平衡的树比较简单，生成一个已排序列表，并从中创建一棵新的树即可。因为要按照升序来创建树，所以你需要按降序来创建一个列表，然后开始左旋结果。当然你也可以选择对称的方式。

以下是获得完全平衡树的算法：

- 1 向左旋转树，直到结果的分支尽量相等为止。这意味着如果总大小为奇数，则分支大小将相等；如果总大小为偶数，则分支大小将差 1。结果将是一棵具有两个大小几乎相等的完全不平衡的分支的树。
- 2 在右分支应用同样的递归过程。在左分支应用对称过程（向右旋转）。
- 3 当结果的高度等于 $\log_2(\text{size})$ 时停止。为此，你需要以下辅助方法：

```
public static int log2nlz(int n) {
    return n == 0
        ? 0
        : 31 - Integer.numberOfLeadingZeros(n);
}
```

练习 10.14

实现 `balance` 方法用于完全平衡任意树。它是一个静态方法，以待平衡的树为参数。

提示

这个实现将基于几个辅助方法：第一个方法将通过调用 `toListInOrderRight` 方法来创建完全不平衡的树。结果列表将向左折叠为一棵（完全不平衡的）树，这样会让平衡过程更容易一些。

还需要一个方法来测试树是否完全平衡，还有一个方法用于递归地旋转树。以下是旋转树的方法：

```
public static <A> A unfold(A a, Function<A, Result<A>> f) {
    Result<A> ra = Result.success(a);
    return unfold(new Tuple<>(ra, ra), f).eval()._2.getOrElse(a);
}

private static <A> TailCall<Tuple<Result<A>, Result<A>>> unfold(Tuple<Result<A>,
    Result<A>> a, Function<A, Result<A>> f) {
    Result<A> x = a._2.flatMap(f::apply);
    return x.isSuccess()
        ? TailCall.sus(() -> unfold(new Tuple<>(a._2, x), f))
        : TailCall.ret(a);
}
```

这个方法与 `List.unfold` 或 `Stream.unfold` 类似，称为 `unfold`。它做的任务相同（除了函数的结果类型与其参数类型相同以外），但忽略了结果，而只保留最后两个，所以它更快并且用的内存更少。

答案 10.14

首先定义验证树是否不平衡的工具方法。为了使其平衡，如果分支的总大小是偶数，两个分支的高度差必须为 0；如果是奇数，则为 1：

```
static <A extends Comparable<A>> boolean isUnBalanced(Tree<A> tree) {
    return Math.abs(tree.left().height() - tree.right().height())
           > (tree.size() - 1) % 2;
}
```

接下来就可以编写主要的平衡方法了：

```
public static <A extends Comparable<A>> Tree<A> balance(Tree<A> tree) {
    return balance_(tree.toListInOrderRight().foldLeft(Tree.<A>empty(),
        t -> a -> new T<>(empty(), a, t)));
}

public static <A extends Comparable<A>> Tree<A> balance_(Tree<A> tree) {
    return !tree.isEmpty() && tree.height() > log2nlz(tree.size())
        ? Math.abs(tree.left().height() - tree.right().height()) > 1
          ? balance_(balanceFirstLevel(tree))
            : new T<>(balance_(tree.left()), tree.value(),
                balance_(tree.right()))
          : tree;
}

private static <A extends Comparable<A>> Tree<A>
    balanceFirstLevel(Tree<A> tree) {
    return unfold(tree, t -> isUnBalanced(t)
        ? tree.right().height() > tree.left().height()
          ? Result.success(t.rotateLeft())
          : Result.success(t.rotateRight())
        : Result.empty());
}
```

10.7.3 自动平衡树

尽管 `balance` 方法被设计为在处理大的不平衡树时避免栈溢出，但是你不能在这样的树上使用它，因为它会在平衡过程中栈溢出，这可以在测试中看到。不可能用超过 15 000 个元素的完全不平衡的树来测试 `balance` 方法。

解决办法是仅在小的不平衡树和任意大小的部分平衡的树上使用 `balance` 方法。这意味着你必须在树变得太大之前平衡它。问题在于你是否可以在每次修改后自动进行平衡。

练习 10.15

转换你已开发的树，使其在插入、合并和删除时自动平衡。

答案 10.15

在修改树的每个操作之后调用 `balance` 方法是一个显而易见的方案，如以下

代码所示：

```
@Override
public Tree<A> insert(A a) {
    return balance(ins(a));
}

protected Tree<A> ins(A a) {
    return a.compareTo(this.value) < 0
        ? new T<>(left.ins(a), this.value, right)
        : a.compareTo(this.value) > 0
        ? new T<>(left, this.value, right.ins(a))
        : new T<>(this.left, value, this.right);
}
```

这样做适用于小的树（实际上，不需要平衡），但是对于大的树来说，这样不行，因为它太慢了。有一个方案是并不总平衡树。例如，只有当高度达到完全平衡树理想高度的 20 倍时，才运行平衡方法：

```
public Tree<A> insert(A a) {
    Tree<A> t = ins(a);
    return t.height() > log2nlz(t.size()) * 20 ? balance(t) : t;
}
```

10.7.4 解决正确的问题

平衡方案的性能可能看起来远非最佳，但这是一个权衡。将具有 100 000 个元素的有序列表创建为一棵树将需要 7.5 秒，并生成一棵高度为 59 的树，与此相比，理想高度为 16。在 insert 方法中用 10 替换 20 将会消耗双倍的时间，而没有带来任何好处，因为结果树的高度为 159。请注意，结果的高度与你使用的值并不成正比。如果平衡树的操作离最后一次插入比较近，那就会好得多，所以最好用一个高一点的值，只是为了避免栈溢出并在使用树之前显式地平衡它。

但真正的问题是，你要解决什么问题？事实上，至少有两个非常不同的需求：

- 需要在不冒着栈溢出的风险下，能以任意顺序从大量元素创建树。
- 需要使树尽可能平衡，因为这可以将高度最小化，而搜索所需的时间与高度成正比。

对于第一个要求而言，你无须使树完全平衡。可以接受高度为 2000，因为它不会溢出栈。你可以简单地在每当插入 2000 个元素时平衡树。当构造完成时，再次

平衡树。

第二个要求则是另一回事了，它可能还会有各种不同的用例。一些树几乎从未更新，而其他树则不断地更新。对于第一种情况，可以在每次改变之后平衡树。对于第二种，最好只有在一定数量的更改之后才更新。无论何种方式，一种优化就是批量地修改树并只在一批完成之后才平衡。你将在第 11 章中学到更多相关知识。

10.8 总结

- 树是递归的数据结构，它的一个元素链接到一棵或多棵子树。
- 二叉搜索树可以快速检索可比较的元素。
- 树可以有不同程度的平衡。完全平衡的树提供最佳的性能，而完全不平衡的树的性能与列表相同。
- 树的大小是它所包含的元素数量；树的高度是树中最长的路径。
- 树的结构取决于其元素插入的顺序。
- 可以通过许多不同的顺序（前序、中序或后序）和两个方向（从左到右，或从右到左）来遍历树。
- 树无须遍历就可以很容易地被合并。
- 可以用多种方式对树进行映射、旋转和折叠。
- 树可以通过平衡以获得更佳性能，并避免在递归操作中溢出栈。

用高级树来解决真实问题

本章要点

- 用自平衡树避免栈溢出
- 实现红黑树
- 创建函数式的 map
- 设计一个函数式优先队列

在第 10 章中，你学习了二叉树的结构和树的基本操作。但是，你也看到了要从树中完全获益，必须有非常具体的用例，例如处理随机顺序的数据；或是有限的数据集，以免冒着栈溢出的风险。树的栈安全比列表要难得多，因为每个计算步骤会涉及两个递归调用，这使得创建尾递归版本成为不可能。

在本章中，我们将研究两种具体的树：

- 红黑树（red-black tree）是一种自平衡的通用高性能树。它适用于任何大小的数据集和常用场景。
- 左倾堆（leftist heap）是一种特别适用于实现优先队列的树。

11.1 性能更好且栈安全的自平衡树

在第 10 章中使用的 DSW 平衡算法不太适合函数式的平衡树，因为它是为直接修改而设计的。在函数式编程中，通常会避免直接修改，而用为每个变更创建一个新的结构来替代。一个更好的方案是定义一个平衡过程，它不需要在重建完全不平衡的树并最终平衡它之前将树转换为列表。有两种方法可以优化这个过程：

- 直接旋转原来的树（剔除列表 / 不平衡树的处理）。
- 接受一定量的不平衡。

你可以试着发明一个这样的方案，虽然别人一直都在这么做。红黑树是最高效的自平衡树设计之一。这种结构是 1978 年由 Guibas 和 Sedgewick 发明的。¹在 1999 年，Chris Okasaki 在他的书《纯函数式数据结构》（剑桥大学出版社，1999）中发表了一个函数式版本的红黑树算法。这一描述通过 Standard ML 的实现来阐明，之后又增加了 Haskell 的实现。它就是你将在 Java 中实现的算法。

如果你对函数式数据结构感兴趣，我强烈建议你购买并阅读 Okasaki 的书。你也可以阅读他 1996 年的同名论文。虽然他的书要完整得多，但论文可以免费下载（www.cs.cmu.edu/~rwh/theses/okasaki.pdf）。

11.1.1 树的基本结构

红黑树是二叉搜索树（BST），有一些结构上的补充和一个可以平衡结果的修改版插入算法。不幸的是，Okasaki 没有描述删除，这是一个更加复杂的过程。但是 Kimball Germane 和 Matthew Might 在 2014 年描述了这个“遗漏的方法”。²

在红黑树中，每棵树（包括子树）都有一个表示其颜色的附加属性。除此以外，它的结构与 BST 结构完全相同，如清单 11.1 所示。

¹ Leo J. Guibas 和 Robert Sedgewick, “A dichromatic framework for balanced trees（平衡树的双色框架）”, *Foundations of Computer Science* (1978), <http://mng.bz/Ly5Jl>.

² Kimball Germane 和 Matthew Might, “Functional Pearl, Deletion: The curse of the red-black tree（函数式明珠，删除：红黑树的诅咒）”, *JFP* 24,4 (2014): 423-433; <http://matt.might.net/papers/germane2014deletion.pdf>.

清单 11.1 红黑树的基本结构

```

public abstract class Tree<A extends Comparable<A>> {

    private static Tree E = new E();
    private static Color R = new Red();
    private static Color B = new Black();
    protected abstract boolean isE();
    protected abstract boolean isT();
    protected abstract boolean isB();
    protected abstract boolean isR();
    protected abstract boolean isTB();
    protected abstract boolean isTR();
    public abstract boolean isEmpty();
    protected abstract Tree<A> right();
    protected abstract Tree<A> left();
    protected abstract A value();
    public abstract int size();
    public abstract int height();

    private static class E<A extends Comparable<A>> extends Tree<A> {

        @Override
        protected boolean isE() {
            return true;
        }

        @Override
        public int size() {
            return 0;
        }

        @Override
        public int height() {
            return -1;
        }

        @Override
        public Tree<A> right() {
            return E;
        }

        @Override
        public Tree<A> left() {
            return E;
        }

        @Override
        protected A value() {
            throw new IllegalStateException("value called on Empty");
        }
    }
}

```

颜色用静态单例表示

isE 方法（现在）只是 isEmpty 的快捷方式

定义方法来检查树的每个特征（是否为空、颜色和它们的一些组合）

空类命名为 E。仅仅是为了方便而已

```

@Override
protected boolean isR() {
    return false;
}

@Override
protected boolean isT() {
    return false;
}

@Override
protected boolean isB() {
    return true;
}

@Override
protected boolean isTB() {
    return false;
}

@Override
protected boolean isTR() {
    return false;
}

@Override
public boolean isEmpty() {
    return true;
}

@Override
public String toString() {
    return "E";
}
}

private static class T<A> extends Comparable<A>> extends Tree<A> {

    private final Tree<A> left;
    private final Tree<A> right;
    private final A value;
    private final Color color;
    private final int length;
    private final int height;

    private T(Color color, Tree<A> left, A value, Tree<A> right) {
        this.color = color;
        this.left = left;
        this.right = right;
        this.value = value;
    }
}

```

一棵空树总是黑色的

用一种颜色构建非空树

```
this.length = left.size() + 1 + right.size();
this.height = Math.max(left.height(), right.height()) + 1;
}

public boolean isR() {
    return this.color.isR();
}

public boolean isB() {
    return this.color.isB();
}

@Override
protected boolean isTB() {
    return this.color.isB();
}

@Override
protected boolean isTR() {
    return this.color.isR();
}

@Override
protected boolean isE() {
    return false;
}

@Override
protected boolean isT() {
    return true;
}

@Override
public int size() {
    return length;
}

@Override
public int height() {
    return height;
}

@Override
public boolean isEmpty() {
    return false;
}

@Override
protected Tree<A> right() {
    return right;
}
```

```

@Override
protected Tree<A> left() {
    return left;
}

@Override
protected A value() {
    return value;
}

@Override
public String toString() {
    return String.format("(T %s %s %s %s)", color, left, value, right);
}
}

private static abstract class Color {
    abstract boolean isR();
    abstract boolean isB();
}

private static class Red extends Color {

    @Override
    boolean isR() {
        return true;
    }

    @Override
    boolean isB() {
        return false;
    }

    @Override
    public String toString() {
        return "R";
    }
}

private static class Black extends Color {

    @Override
    boolean isR() {
        return false;
    }

    @Override
    boolean isB() {
        return true;
    }
}

```

Red 和 Black 颜色类继承了 Color 抽象类

```
@Override
public String toString() {
    return "B";
}

public static <A extends Comparable<A>> Tree<A> empty() {
    return E;
}
```

没有体现 member 方法和诸如 fold、map 等其他方法，因为它们与树的标准版本没有区别。如你所见，只有插入和删除方法不同。

11.1.2 往红黑树中插入元素

红黑树的主要特征是总要验证不变量。在修改树时，将进行测试以检查是否破坏了这些不变量，并在必要时通过旋转和变更颜色来恢复它们。这些不变量如下：

- 一棵空树是黑色的。（它是不可变的，因此没有必要验证。）
- 一棵红树的左右子树是黑色的。换句话说，在向下查找树时，不可能找到两个连续的红色。
- 从根到空子树的每条路径都有数量相同的黑色。

往红黑树中插入一个元素是一个复杂的过程，包括在插入后检查不变量（如有必要，重新平衡）。相应的算法如下：

- 一棵空树总是黑色的。
- 与普通树的插入完全一样，但接下来要平衡。
- 往空树中插入一个元素以生成一棵红树。
- 平衡之后，把根变为黑色。

图 11.1 到图 11.7 阐明了往一棵开始为空的树中插入整数 1 到 7 的过程。图 11.1 展示了往空树中插入元素 1。由于你插入了一棵空树，所以初始颜色为红。插入元素后，把根变为黑色。

从一棵空树开始

插入1

红色，因为它被插入到一棵空树中



把根变为黑色

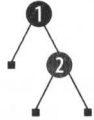


图 11.1 往一开始为空的树中插入整数 1 到 7，步骤 1。

图 11.2 展示了插入元素 2。插入的元素为红色，根已经是黑色的了，所以没有平衡的必要。

插入2

根已为黑色



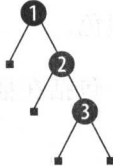
平衡：什么也不用做

图 11.2 往一开始为空的树中插入整数 1 到 7，步骤 2。

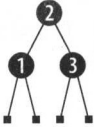
图 11.3 展示了插入元素 3。插入的元素为红色，因为这棵树有两个连续的元素为红色，所以它需要平衡。由于红色元素现在有两个子元素，把它们变为黑色。（红元素的子元素总是黑色的）最后，把根变为黑色。

插入3

红色，因为它被插入到一棵空树中



平衡



把根变为黑色

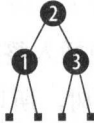


图 11.3 往一开始为空的树中插入整数 1 到 7，步骤 3。

图 11.4 展示了插入元素 4。不需要进一步操作。

插入4

平衡并把根变为黑色：什么也不用改

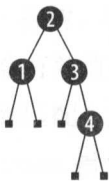


图 11.4 往一开始为空的树中插入整数 1 到 7，步骤 4。

图 11.5 阐明了插入元素 5。你现在有两个连续红色元素，所以必须通过把 3 变为 4 的左子元素来平衡该树。而 4 则变为 2 的右子元素。

插入5

平衡

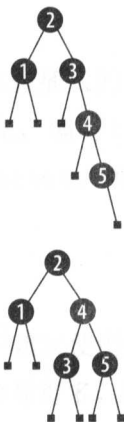


图 11.5 往一开始为空的树中插入整数 1 到 7，步骤 5。

图 11.6 展示了插入元素 6。不需要进一步操作。

插入6

平衡：什么也不用做

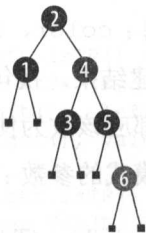
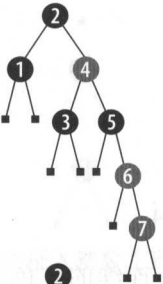


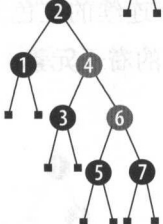
图 11.6 往一开始为空的树中插入整数 1 到 7，步骤 6。

在图 11.7 中，元素 7 被添加到树中。因为元素 6 和 7 是两个连续红色元素，所以树需要平衡。第一步是把 5 变为 6 的左子元素，并且把 6 变为 4 的右子元素，这样又出现了两个连续红色元素：4 和 6。然后再次平衡该树，以 4 为根，2 变为 4 的左子元素，3 变为 2 的右子元素。最后的操作也包括了把根变为黑色。

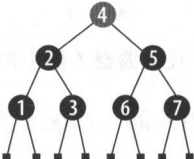
插入7



平衡



平衡



把根变为黑色

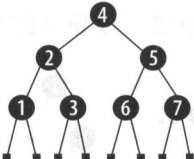


图 11.7 往一开始为空的树中插入整数 1 到 7，步骤 7。

balance 方法的参数与树的构造函数相同：color、left、value 和 right。这 4 个参数检查了各种模式，都可以相应地构建结果。换句话说，balance 方法取代了树的构造函数。任何使用构造函数的程序都应该改为使用该方法。

以下清单展示了如何通过该方法转换每个模式的参数：

- $(TB(TR(TR\ a\ x\ b)\ y\ c)\ z\ d) \rightarrow (TR(TB\ a\ x\ b)\ y\ (TB\ c\ z\ d))$
- $(TB(TR\ a\ x\ (TR\ b\ y\ c))\ z\ d) \rightarrow (TR(TB\ a\ x\ b)\ y\ (TB\ c\ z\ d))$
- $(TB\ a\ x\ (TR(TR\ b\ y\ c)\ z\ d)) \rightarrow (TR(TB\ a\ x\ b)\ y\ (TB\ c\ z\ d))$
- $(TB\ a\ x\ (TR\ b\ y\ (TR\ c\ z\ d))) \rightarrow (TR(TB\ a\ x\ b)\ y\ (TB\ c\ z\ d))$
- $(T\ color\ a\ x\ b) \rightarrow (T\ color\ a\ x\ b)$

每一对括号都对应着一棵树。字母 T 表示非空树。B (黑) 和 R (红) 表示颜色。小写字母是相应位置上任意有效值的占位符。每个左边的模式 (→ 箭头的左边) 都是降序, 这意味着如果找到匹配, 则应用右边的相应模式作为结果树。这种展示方式非常类似于 switch ... case 指令, 最后一行表示默认情况。

练习 11.1

编写 insert、balance 和 blacken 方法来实现红黑树的插入。不幸的是, Java 没有实现模式匹配, 所以你不得不使用条件指令来代替。

提示

编写一个执行常规插入的 ins 方法, 然后把构造函数的调用替换为对 balance 方法的调用。接下来, 编写 blacken 方法, 最后在父类中编写 insert 方法, 用 ins 的结果调用 blacken。所有这些方法都应该是 protected, 只有 insert 方法是 public。

答案 11.1

仅此一次, 我不建议使用条件运算符。用一个连续且都包含一个 return 的 if 来表示这些模式要容易得多。balance 方法如下:

```
Tree<A> balance(Color color, Tree<A> left, A value, Tree<A> right) {
    if (color.isB() && left.isTR() && left.left().isTR()) {
        return new T<>(R, new T<>(B, left.left().left(), left.left().value(),
            left.left().right(), left.value(), new T<>(B, left.right(), value,
            right)));
    }
    if (color.isB() && left.isTR() && left.right().isTR()) {
        return new T<>(R, new T<>(B, left.left(), left.value(),
            left.right().left(), left.right().value(), new T<>(B,
            left.right().right(), value, right)));
    }
    if (color.isB() && right.isTR() && right.left().isTR()) {
        return new T<>(R, new T<>(B, left, value, right.left().left(),
            right.left().value(), new T<>(B, right.left().right(),
            right.value(), right.right()));
    }
    if (color.isB() && right.isTR() && right.right().isTR()) {
        return new T<>(R, new T<>(B, left, value, right.left(), right.value(),
            new T<>(B, right.right().left(), right.right().value(),
            right.right().right()));
    }
    return new T<>(color, left, value, right);
}
```

每个 if 都实现了本练习前面列出的一个模式。如果你想比较它们，在文本编辑器中要比印刷的这页书容易得多。

ins 方法与标准 BST 中的方法非常相似，只是 balance 方法替换了 T 的构造函数（还有额外的颜色参数）。T 类的实现如下：

```
protected Tree<A> ins(A value) {
    return value.compareTo(this.value) < 0
        ? balance(this.color, this.left.ins(value), this.value, this.right)
        : value.compareTo(this.value) > 0
            ? balance(this.color, this.left, this.value,
                    this.right.ins(value))
            : this;
}
```

还有 E 类的实现如下：

```
protected Tree<A> ins(A value) {
    return new T<>(R, empty(), value, empty());
}
```

在 Tree 类中实现 blacken 方法：

```
protected static <A extends Comparable<A>> Tree<A> blacken(Tree<A> t) {
    return t.isEmpty()
        ? empty()
        : new T<>(B, t.left(), t.value(), t.right());
}
```

最后，在 Tree 类中定义了 insert 方法，并返回把 ins 变为黑色的结果：

```
public Tree<A> insert(A value) {
    return blacken(ins(value));
}
```

从红黑树中删除元素 Kimball Germane 和 Matthew Might 在一篇题为

The missing method: Deleting from Okasaki's red-black trees (遗漏的方法：

从 Okasaki 的红黑树中删除) ([http://matt.might.net/articles/red-black-](http://matt.might.net/articles/red-black-delete/)

delete/) 的文章中讨论了从红黑树中删除元素。Java 中的实现太长了，

难以在本书中囊括，但是它包含在随书附带的代码中 ([http://github.](http://github.com/fpinjava/fpinjava)

com/fpinjava/fpinjava)。下一个练习将会用到。

11.2 红黑树的用例：map

整型树并不总是有用的（尽管有时它们是）。二叉搜索树的一个非常重要的用途是 map，也称为字典（dictionary）或关联数组（associative array）。map 是一个允许插入、删除和快速检索每个键 / 值对的集合。Java 程序员很熟悉 map，Java 提供了几个实现，其中最常用的是 HashMap 和 TreeMap。然而，若不使用难以正确设计与使用的某些保护机制（虽然可以用它们的并发版本），这些 map 不能用于多线程环境。

11.2.1 实现 map

函数式的树，如你开发的红黑树，具有不变性的优点，允许你将它们用于多线程环境，而不用担心锁与同步。清单 11.2 展示了可以用红黑树实现的 Map 接口。

清单 11.2 一个函数式的 map

```
public class Map<K extends Comparable<K>, V> {  
    public Map<K, V> add(K key, V value) {  
        ...  
    }  
  
    public boolean contains(K key) {  
        ...  
    }  
  
    public Map<K, V> remove(K key) {  
        ...  
    }  
  
    public Result<MapEntry<K, V>> get(K key) {  
        ...  
    }  
  
    public boolean isEmpty() {  
        ...  
    }  
  
    public static <K extends Comparable<K>, V> Map<K, V> empty() {  
        return new Map<>();  
    }  
}
```

练习 11.2

实现所有方法以完成 Map 类。

提示

你应该使用一个委托。通过这个委托，所有的方法都可以用一行代码实现。唯一（非常容易）的问题是决定如何在 map 中存储数据。

答案 11.2

答案是创建一个组件来表示键 / 值对，并将该组件的实例存储在树中。这个组件与 Tuple 非常相似，但是有一个重要的区别：它必须能基于 key 来进行比较。equals 和 hashCode 方法也将基于键的相等性和散列码。一个可能的实现如下所示：

```
public class MapEntry<K extends Comparable<K>, V>
    implements Comparable<MapEntry<K, V>> {
    public final K key;
    public final Result<V> value;

    private MapEntry(K key, Result<V> value) {
        this.key = key;
        this.value = value;
    }

    @Override
    public String toString() {
        return String.format("MapEntry(%s, %s)", key, value);
    }

    @Override
    public int compareTo(MapEntry<K, V> me) {
        return this.key.compareTo(me.key);
    }

    @Override
    public boolean equals(Object o) {
        return o instanceof MapEntry && this.key.equals(((MapEntry) o).key);
    }

    @Override
    public int hashCode() {
        return key.hashCode();
    }

    public static <K extends Comparable<K>, V> MapEntry<K, V>
        mapEntry(K key, V value) {
        return new MapEntry<>(key, Result.success(value));
    }
}
```

```

    }
    public static <K extends Comparable<K>, V> MapEntry<K, V>
                                   mapEntry(K key) {
        return new MapEntry<>(key, Result.empty());
    }
}

```

现在实现 Map 组件只需把所有的操作委托给一个 Tree<MapEntry<Key, Value>> 即可。一个可能的实现如下所示：

```

import static com.fpinjava.advancedtrees.exercisell_02.MapEntry.*;

public class Map<K extends Comparable<K>, V> {

    protected final Tree<MapEntry<K, V>> delegate;

    private Map() {
        this.delegate = Tree.empty();
    }

    private Map(Tree<MapEntry<K, V>> delegate) {
        this.delegate = delegate;
    }

    public Map<K, V> add(K key, V value) {
        return new Map<>(delegate.insert(mapEntry(key, value)));
    }

    public boolean contains(K key) {
        return delegate.member(mapEntry(key));
    }

    public Map<K, V> remove(K key) {
        return new Map<>(delegate.delete(mapEntry(key)));
    }

    public MapEntry<K, V> max() {
        return delegate.max();
    }

    public MapEntry<K, V> min() {
        return delegate.min();
    }

    public Result<MapEntry<K, V>> get(K key) {
        return delegate.get(mapEntry(key));
    }

    public boolean isEmpty() {
        return delegate.isEmpty();
    }
}

```

```

    }
    public static <K extends Comparable<K>, V> Map<K, V> empty() {
        return new Map<>();
    }
}

```

11.2.2 扩展 map

由于某些操作在当前情况下没有任何意义，所以并非所有的树操作都被委托。但在某些特殊情况下你可能会需要额外的操作。实现这些操作很简单：扩展 Map 类并增加委托方法。例如，你可能需要找到最大键或最小键的对象。另一个可能需要的是折叠 map，也许可用于获取一个包含所有值的列表。以下是一个委托 foldLeft 方法的示例：

```

public <B> B foldLeft(B identity, Function<B,
    Function<MapEntry<K, V>, B>> f, Function<B, Function<B, B>>> g) {
    return delegate.foldLeft(identity, b -> me -> f.apply(b).apply(me), g);
}

```

通常，折叠 map 发生在非常特定的用例中，值得在 Map 类中进行抽象。

练习 11.3

在 Map 类中编写一个 values 方法，按键的升序返回 map 中包含的值列表。

提示

你可能需要在 Tree 类中创建一个新的折叠方法，并在 Map 类中委托给它。

答案 11.3

values 方法有几种可能的实现。可以委托 foldInOrder 方法，但是这个方法按升序来迭代树的值。用这个方法构造列表将导致列表降序排列。你可以反转结果，但效率不会太高。

一个更好的方案是在 Tree 类中增加一个 foldInReverseOrder 方法。回想一下 foldInOrder 方法：

```

public <B> B foldInOrder(B identity,
    Function<B, Function<A, Function<B, B>>> f) {
    return f.apply(left.foldInOrder(identity, f))
        .apply(value)
}

```

```
.apply(right.foldInOrder(identity, f));
```

你需要做的全部工作只是反转顺序而已：

```
public <B> B foldInReverseOrder(B identity,
    Function<B, Function<A, Function<B, B>>> f) {
    return f.apply(right.foldInReverseOrder(identity, f))
        .apply(value).apply(left
            .foldInReverseOrder(identity, f));
}
```

Empty 的实现照常返回 identity。现在，你可以从 Map 类的内部委托到这个方法：

```
public List<V> values() {
    return List.sequence(delegate.foldInReverseOrder(List.<Result<V>>list(),
        lst1 -> me -> lst2 -> List.concat(lst2,
            lst1.cons(me.value))))).getOrElse(List.list());
}
```

如果有类型的问题，你可以用显式类型来编写这个函数：

```
Function<List<Result<V>>, Function<MapEntry<K, V>,
    Function<List<Result<V>>, List<Result<V>>>>> f =
    lst1 -> me -> lst2 -> List.concat(lst2, lst1.cons(me.value));
```

11.2.3 使用键不可比较的 map

Map 类不仅有用而且相对高效，但与你可能已经习惯使用的 map 相比，它具有很大的缺点：键必须是可比较的。用于键的类型一般是可比较的，例如整型或字符串，但如果你需要使用不可比较类型的键呢？

练习 11.4

实现一个 Map 的版本，其键不可比较。

提示

有两件事需要修改。首先，虽然键不可比较，但是 MapEntry 类应该是可比较的。其次，相等的 map 条目可能会保存不相等的值，因此应该通过保留所有的冲突条目来解决冲突。

答案 11.4

首先要做的就是修改 `MapEntry` 类，删除键可以比较的说明：

```
public class MapEntry<K, V> implements Comparable<MapEntry<K, V>> {
```

请注意，`MapEntry` 类仍然是可以比较的，尽管 `K` 类型已经不是了。

其次，你需要对 `compareTo` 方法使用不同的实现。办法之一是通过基于键的散列码来比较 `map` 条目：

```
public int compareTo(MapEntry<K, V> that) {

    int thisHashCode = this.hashCode();
    int thatHashCode = that.hashCode();

    return thisHashCode < thatHashCode
        ? -1
        : thisHashCode > thatHashCode
        ? 1
        : 0;
}
```

其次你需要处理发生在两个 `map` 条目之间的冲突，它们具有不同的键和相同的散列码。在这种情况下，你应该两者都保留。最简单的解决方案是将 `map` 条目存储在列表中，你需要为此修改 `Map` 类。

首先，修改委托树的类型：

```
protected final Tree<MapEntry<Integer, List<Tuple<K, V>>>> delegate;
```

然后需要把构造函数更改为以委托为参数：

```
public Map(Tree<MapEntry<Integer, List<Tuple<K, V>>>> delegate) {
    this.delegate = delegate;
}
```

接下来，你需要一个方法来获取键的相同散列码所对应的键 / 值元组列表：

```
private Result<List<Tuple<K, V>>> getAll(K key) {
    return delegate.get(mapEntry(key.hashCode()))
        .flatMap(x -> x.value.map(lt -> lt.map(t -> t)));
}
```

然后你可以通过 `getAll` 方法来定义 `add`、`contains`、`remove` 和 `get` 方法。
`add` 方法如下：

```

public Map<K, V> add(K key, V value) {
    Tuple<K, V> tuple = new Tuple<>(key, value);
    List<Tuple<K, V>> ltkv = getAll(key).map(lt ->
        lt.foldLeft(List.list(tuple), l -> t -> t._1.equals(key)
            ? l
            : l.cons(t))).getOrElse(() -> List.list(tuple));
    return new Map<>(delegate.insert(mapEntry(key.hashCode(), ltkv)));
}

```

contains 方法如下:

```

public boolean contains(K key) {
    return getAll(key).map(lt -> lt.exists(t ->
        t._1.equals(key))).getOrElse(false);
}

```

remove 方法如下:

```

public Map<K, V> remove(K key) {
    List<Tuple<K, V>> ltkv = getAll(key).map(lt ->
        lt.foldLeft(List.<Tuple<K, V>>list(), l -> t -> t._1.equals(key)
            ? l
            : l.cons(t))).getOrElse(List::list);
    return ltkv.isEmpty()
        ? new Map<>(delegate.delete(MapEntry.mapEntry(key.hashCode())))
        : new Map<>(delegate.insert(mapEntry(key.hashCode(), ltkv)));
}

```

```

public Result<Tuple<K, V>> get(K key) {
    return getAll(key).flatMap(lt -> lt.first(t -> t._1.equals(key)));
}

```

最后, 需要删除 min 和 max 方法。

通过这些修改, Map 类可以用于不可比较的键。用列表存储键 / 值元组可能不是最高效的实现, 因为搜索列表需要的时间与元素的数量成比例。但是在大多数情况下, 列表中只包含一个元素, 因此搜索将会立即返回。

值得注意的是, 这个实现的 remove 方法会检查元组的结果列表是否为空。如果是, 它就调用委托的 delete 方法。否则, 它就调用 insert 方法来重新插入相应条目已被删除的新列表。回想一下练习 10.1。可以这么做只是因为插入是这样实现的: 当要插入的元素与 map 中已存在的元素相等时, 将其插入以替换原来的元素。如果你先前没有这样实现, 就不得不首先删除元素, 然后将新的元素插入已修改的列表中。

11.3 实现函数式优先队列

如你所知，队列是一种具有特定访问协议的列表。队列可以只有一端，就像在前几章中经常使用的单链表一样。在这种情况下，访问协议为后进先出（LIFO）。队列也可以有两端，允许先进先出（FIFO）的访问协议。但是也有协议更加特定的数据结构，优先队列是其中一员。

11.3.1 优先队列访问协议

可以将值以任何顺序插入优先队列中，但只能以指定的顺序获取。所有的值都有优先级，只有优先级最高的元素才可用。优先级表示为元素的顺序，这意味着元素必须有某种可以比较的方式。

优先级在理论上的等待队列中对应于元素的位置。位置最低的元素（第一个元素）拥有最高的优先级。因此按照惯例，最小值代表最高的优先级。

由于优先队列会包含可比较的元素，这使得它非常适合树状结构。但是从用户的角度上看，优先队列被视为一个列表，其中包含一个 head（优先级最高的元素，即最小值）和一个 tail（队列的剩余部分）。

11.3.2 优先队列使用案例

优先队列有许多不同的用例。首先映入脑海中的就是排序。你可以将元素随机插入优先队列，并获得排序后的结果。这并不是这个结构的主要用例，但它对于排序小数据集不无裨益。

另一个非常常见的用例是异步并行处理后重新排序元素。假设你有多页数据要处理。为了加快处理速度，你可以将数据分发给几个并行工作的线程。但是不能保证线程会按照它们收到的顺序返回工作结果。为了重新同步数据页，你可以将它们放在优先队列中。使用该页的进程会在之后轮询队列以检查可用元素（队列的 head）是否是预期的元素。例如，如果页 1、2、3、4、5、6、7 和 8 分配给了要并行处理的 8 个线程，则消费者将轮询队列以查看页 1 是否可用。如果可用，那就消费它。否则就等待一下。

在这种场景下，队列既是缓冲，又是重新排序元素的一种方式。这通常意味着大小上的变化有限，因为从队列中删除元素的速度会与插入多少有几分相同。当然，

如果消费者消费元素的速度与 8 个线程的生产速度大致相同，确是如此。否则，也许可以用几个消费者。

正如我前面所说，选择一个实现通常是以空间换时间或者以时间换时间的权衡。在这里，你必须在插入和获取时间之间做出选择。在一般用例中，优化获取时间要优先于插入时间，因为就插入和获取这两个操作数量之比而言，一般来说获取都要大得多。（经常读取 `head` 但不会删除。）

11.3.3 实现需求

你可以基于红黑树来实现优先队列，因为很快便可以找到最小值。但获取并不意味着删除。如果搜索最小值并发现它不是你想要的，你会稍后再回来搜索。解决这个问题一个办法是记忆插入的最小值。你可能想做的其他变更是删除。删除元素相对较快，但由于总是删除最小的元素，所以应该能够优化这个操作的数据结构。

另一个重要的问题与重复有关。虽然红黑树不允许重复，但优先队列必须允许，因为完全可能有几个优先级相同的元素。解决方案可以与 `map` 相同——存储优先级相同的元素列表（而不是单个元素）——但是性能上可能不够好。

11.3.4 左倾堆数据结构

为了满足对优先队列的需求，你将使用 Okasaki 在他的书 *Purely Functional Data Structures* 中描述的“左倾堆”。¹ 该数据结构满足优先队列的需求。Okasaki 将左倾堆定义为“具有附加左倾属性的堆排序树”：

- 堆排序树是一棵其元素的每个分支都大于等于元素本身的树。这样保证了树中最小的元素始终是根元素，允许即时访问最小值。
- “左倾”属性意味着，对于每个元素，左分支的等级（`rank`）都大于等于右分支的等级。
- 元素的等级是沿右侧路径（也称为右脊，`right spine`）到空元素的长度。左倾属性保证了从任何元素到空元素的最短路径都是右路径。结果就是元素总是沿着任意下降的路径按升序排列。

¹ Clark Allan Crane 首先在 *Linear lists and priority queues as balanced binary trees*（以线性列表和优先队列作为平衡二叉树）（1972）中描述了左倾堆，但是 Okasaki 是第一个发布纯函数式实现的人之一。

图 11.8 展示了左偏树（leftist tree，即左倾堆）的一个示例。

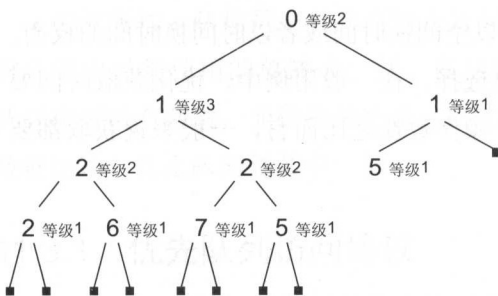


图 11.8 一个堆排序左偏树，展示了元素的每个分支都大于等于元素本身，每个左分支的等级都大于等于相应的右分支的等级。

如你所见，能够以常量时间查询优先级最高的元素，因为它永远是树的根。该元素被称为结构的“head”。删除一个元素与列表类似，一旦根被删除，将返回树的剩余部分。这个返回值被称为结构的“tail”。

11.3.5 实现左倾堆

左倾堆的主类称为 Heap，并实现为一棵树。基本结构如清单 11.3 所示。与先前发展至今的树的主要区别在于，right、left 和 head（等价于先前示例中称为 value 的方法）将返回一个 Result 而非原始值。还要注意的，元素的数量称为 length（与队列类似），并且记忆 length 和 rank 将由构造函数的调用者而不是构造函数本身计算。这并非一个有意为之的设计抉择，只是想展示另一种做事的方式。构造函数是私有的，所以差异不会泄漏到 Heap 类以外。

清单 11.3 左倾堆的结构

```
public abstract class Heap<A extends Comparable<A>> {

    @SuppressWarnings("rawtypes")
    protected static final Heap EMPTY = new Empty();
    protected abstract Result<Heap<A>> left();
    protected abstract Result<Heap<A>> right();
    protected abstract int rank();
    public abstract Result<A> head();
    public abstract int length();
    public abstract boolean isEmpty();

    public static class Empty<A extends Comparable<A>> extends Heap<A> {
```

left、right 和 head 方法都
返回一个 Result

树的长度只是它所包
含的元素数量

```

private Empty() {}

@Override
protected int rank() {
    return 0;
}

@Override
public Result<A> head() {
    return Result.failure(new NoSuchElementException(
        "head() called on empty heap"));
}

@Override
public int length() {
    return 0;
}

@Override
protected Result<Heap<A>> left() {
    return Result.success(empty());
}

@Override
protected Result<Heap<A>> right() {
    return Result.success(empty());
}

@Override
public boolean isEmpty() {
    return true;
}
}

public static class H<A> extends Comparable<A>> extends Heap<A> {

    private final int length;
    private final int rank;
    private final A head;
    private final Heap<A> left;
    private final Heap<A> right;

    private H(int length, int rank, Heap<A> left, A head, Heap<A> right) {
        this.length = length;
        this.rank = rank;
        this.head = head;
        this.left = left;
        this.right = right;
    }

    @Override

```

等级和长度属性在 H 子类以外计算

```

protected int rank() {
    return this.rank;
}

@Override
public Result<A> head() {
    return Result.success(this.head);
}

@Override
public int length() {
    return this.length;
}

@Override
protected Result<Heap<A>> left() {
    return Result.success(this.left);
}

@Override
protected Result<Heap<A>> right() {
    return Result.success(this.right);
}

@Override
public boolean isEmpty() {
    return false;
}
}

@SuppressWarnings("unchecked")
public static <A extends Comparable<A>> Heap<A> empty() {
    return EMPTY;
}
}

```

练习 11.5

要添加到 Heap 实现的第一个功能是添加一个元素，为此定义一个 add 方法。在 Heap 类中创建一个具有如下签名的实例方法：

```
public Heap<T> add(T element)
```

要求是：如果值小于堆中的任何元素，它应该成为新堆的根。否则，堆的根不应该改变。此外，也应遵守对右侧路径的等级和长度的其他要求。

提示

定义一个静态方法，通过一个元素创建一个 Heap；还有另一个方法，通过合并两个堆来创建一个堆，签名如下：

```
public static <A extends Comparable<A>> Heap<A> heap(A element)
public static <A extends Comparable<A>> Heap<A> merge(Heap<A> first,
                                                         Heap<A> second)
```

然后通过这两个方法来定义 add 方法。

答案 11.5

从单个元素创建堆的方法很简单。只需创建一个长度为 1，等级为 1 的新树即可；元素参数作为 head；还有两个空堆作为左右分支：

```
public static <A extends Comparable<A>> Heap<A> heap(A element) {
    return new H<>(1, 1, empty(), element, empty());
}
```

通过合并两个堆来创建一个堆会更加复杂一些。为此你还需要一个额外的辅助方法，可以通过一个元素和两个堆来创建堆：

```
protected static <A extends Comparable<A>> Heap<A> heap(A head,
                                                         Heap<A> first, Heap<A> second) {
    return first.rank() >= second.rank()
        ? new H<>(first.length() + second.length() + 1,
                  second.rank() + 1, first, head, second)
        : new H<>(first.length() + second.length() + 1,
                  first.rank() + 1, second, head, first);
}
```

这段代码首先检查第一个堆的等级是否大于等于第二个堆的等级。如果是，则将新的等级设置为第二个堆的等级 +1，并且依次使用两个堆。否则，将新的等级设置为第一个堆的等级 +1，并且以反序使用两个堆（先是第二，然后第一）。

合并两个堆的方法可以如下编写：

```
public static <A extends Comparable<A>> Heap<A> merge(Heap<A> first,
                                                         Heap<A> second) {
    return first.head().flatMap(
        fh -> second.head().flatMap(
            sh -> fh.compareTo(sh) <= 0
                ? first.left().flatMap(
                    fl -> first.right().map(
                        fr -> heap(fh, fl, merge(fr, second)))
                : first.right().flatMap(
                    fr -> first.left().map(
                        fl -> heap(fh, fr, merge(fl, second)))
        )
    );
}
```



```

: second.left().flatMap(
  sl -> second.right().map(
    sr -> heap(sh, sl, merge(first, sr))))))
    .getOrElse(first.isEmpty() ? second : first);
}

```

当然，如果要合并的一个堆为空，你可以返回另一个堆。否则，计算合并的结果。

如果你发现这段代码难以理解（我希望现在你还不是），它的功能与以下不太函数式的实现完全相同：

```

public static <A extends Comparable<A>> Heap<A> merge(Heap<A> first, Heap<A>
second) {
  return first.isEmpty()
    ? second
    : second.isEmpty()
      ? first
      : first.head().successValue()
        .compareTo(second.head().successValue()) <= 0
        ? heap(first.head().successValue(), first.left()
            .successValue(), merge(first.right()
            .successValue(), second))
        : heap(second.head().successValue(), second.left()
            .successValue(), merge(second.right()
            .successValue(), first));
}

public static <A extends Comparable<A>> Heap<A> merge(Heap<A> first,
Heap<A> second) {
  try {
    return first.head().successValue()
      .compareTo(second.head().successValue()) <= 0
      ? heap(first.head().successValue(), first.left().successValue(),
        merge(first.right().successValue(), second))
      : heap(second.head().successValue(), second.left().successValue(),
        merge(second.right().successValue(), first));
  } catch (IllegalStateException e) {
    return first.isEmpty() ? second : first;
  }
}

```

作为通用法则，你应该始终记住，调用 `successValue` 就像 `getOrThrow`，如果 `Result` 为 `Empty`，可能会抛出异常。你可以先检查空（如上面的第一个例子所示），或是将代码包含在 `try ... catch` 块中（如第二个例子所示），但是这些方案都不是真正的函数式方案。

顺便说一下，你应该尽量避免调用 `successValue` 和 `getOrThrow`。`successValue` 方法只能在 `Result` 类中使用。强制这样操作的最佳方案是使其成

为 `protect`，但在学习时可以用它来查看发生了什么。

随着这些方法定义完毕，可以很容易地创建 `add` 方法了：

```
public Heap<A> add(A element) {
    return merge(this, heap(element));
}
```

11.3.6 实现像队列一样的接口

虽然实现为一棵树，但是从用户的角度看，堆就像是一个优先队列，即 `head` 总为最小元素的一种链表。以此类推，树的根元素被称为 `head`，“删除” `head` 之后的剩余元素称为 `tail`。

练习 11.6

定义一个 `tail` 方法，返回删除 `head` 后的剩余元素。这个方法就像 `head` 方法一样，返回一个 `Result`，以便能够在空队列上安全地调用它。以下是它在 `Heap` 父类中的签名：

```
Result<Heap<A>> tail()
```

答案 11.6

很明显，`Empty` 的实现就是返回一个 `Failure`：

```
public Result<Heap<A>> tail() {
    return Result.failure(new NoSuchElementException("tail() called
                                                    on empty heap"));
}
```

考虑到你在练习 11.5 中定义的方法，`H` 的实现并不复杂。它只是返回左右分支的合并结果：

```
public Result<Heap<A>> tail() {
    return Result.success(Heap.merge(left, right));
}
```

练习 11.7

实现一个接收 `int` 参数的 `get` 方法，并按优先顺序返回第 n 个元素。这个方法将返回一个 `Result` 以处理找不到元素的情况。以下是 `Heap` 父类中的签名：

```
public abstract Result<A> get(int index)
```

答案 11.7

很明显, Empty 的实现就是返回一个 Failure:

```
public Result<A> get(int index) {
    return Result.failure(new NoSuchElementException("Index out of range"));
}
```

H 的实现同样简单。它首先检查索引, 如果为 0, 则返回包含 head 值的一个 Success。否则, 它递归地搜索 tail 中索引为 $n - 1$ 的元素。因为 tail 并不真的存在, 而只是 getTail 方法的返回值 (是一个 Result), 所以用一个对 get 的递归调用来 flatMap 这个结果:

```
public Result<A> get(int index) {
    return index == 0
        ? head()
        : tail().flatMap(x -> x.get(index - 1));
}
```

11.4 元素不可比较的优先队列

要往优先队列中插入元素, 你需要能够比较它们的优先级。但优先级并不总是元素的属性; 并非所有元素都实现了 Comparable 接口。仍然可以使用 Comparator 来比较未实现该接口的元素, 你可以为优先队列实现这个操作吗?

练习 11.8

修改 Heap 类以使用 Comparable 元素或单独的 Comparator。

答案 11.8

首先, 你可以往 Heap 类中添加一个返回 Comparator 的方法。因为比较器是可选的, 所以该方法将返回一个可能为空的 Result<Comparator>。

```
protected abstract Result<Comparator<A>> comparator();
```

然后你可以在两个子类中实现它。Empty 实现将返回在构造函数中初始化的附加属性值:

```
private final Result<Comparator<A>> comparator;
```

```
private Empty(Result<Comparator<A>> comparator) {
    this.comparator = comparator;
}

protected Result<Comparator<A>> comparator() {
    return this.comparator;
}
```

当然,你将在 H 类中也执行相同的操作,不同之处在于你将修改现有的构造函数,而不是创建一个新的构造函数:

```
private final Result<Comparator<A>> comparator;

private H(int length, int rank, Heap<A> left, A head, Heap<A> right,
          Result<Comparator<A>> comparator) {
    this.length = length;
    this.rank = rank;
    this.head = head;
    this.left = left;
    this.right = right;
    this.comparator = comparator;
}

protected Result<Comparator<A>> comparator() {
    return this.comparator;
}
```

然后,你需要更新工厂方法。但是在动手之前,需要改变类的类型参数,把这个

```
public abstract class Heap<A extends Comparable<A>>
```

替换为:

```
public abstract class Heap<A>>
```

应该把相同的修改应用于子类的构造函数。

用于创建空 Heap 的静态工厂方法将会接收一个额外的 Result<Comparator> 参数,你需要添加一个使用默认 Result.Empty 的新方法:

```
public static <A> Heap<A> empty(Comparator<A> comparator) {
    return empty(Result.success(comparator));
}

public static <A> Heap<A> empty(Result<Comparator<A>> comparator) {
    return new Empty<>(comparator);
}
```

请注意，我还添加了一个接收 `Comparator<A>` 而非 `Result<Comparable>` 的方法，以使 `Heap` 类更容易使用。这个方法主要用于 `Heap` 类外部。

尽管如此，你还将保留一个无参的 `empty` 方法。该方法仍然需要使用 `Comparable` 类型参数化。否则，你可能会冒着稍后得到一个 `ClassCastException` 的风险。

```
public static <A extends Comparable<A>> Heap<A> empty() {
    return empty(Result.empty());
}
```

通过使用 `Comparable` 类型，你可以确保得到编译错误而不是运行时异常。

你现在可以对从单个元素创建 `Heap` 的方法同样再来一遍：

```
public static <A extends Comparable<A>> Heap<A> heap(A element) {
    return heap(element, Result.empty());
}

public static <A> Heap<A> heap(A element, Result<Comparator<A>> comparator) {
    Heap<A> empty = empty(comparator);
    return new H<>(1, 1, empty, element, empty, comparator);
}

public static <A> Heap<A> heap(A element, Comparator<A> comparator) {
    Heap<A> empty = empty(comparator);
    return new H<>(1, 1, empty, element, empty, Result.success(comparator));
}
```

接收一个元素和两个堆的方法也需要相应地进行修改，但这次你将从堆参数中获得比较器：

```
protected static <A> Heap<A> heap(A head, Heap<A> first, Heap<A> second) {
    Result<Comparator<A>> comparator = first.comparator()
        .orElse(second::comparator);

    return first.rank() >= second.rank()
        ? new H<>(first.length() + second.length() + 1,
            second.rank() + 1, first, head, second, comparator)
        : new H<>(first.length() + second.length() + 1,
            first.rank() + 1, second, head, first, comparator);
}
```

对于 `merge` 方法，你可以使用任意一个待合并的两棵树的 `Comparator`。如果它们都没有 `Comparator`，你可以使用 `Result.Empty`。为了不从每个递归调用的参数中获得比较器，你可以把方法一分为二：

```
public static <A> Heap<A> merge(Heap<A> first, Heap<A> second) {
    Result<Comparator<A>> comparator =
```

```

        first.comparator().orElse(second::comparator);
        return merge(first, second, comparator);
    }

    public static <A> Heap<A> merge(Heap<A> first, Heap<A> second,
                                   Result<Comparator<A>> comparator) {
        return first.head().flatMap(fh -> second.head()
            .flatMap(sh -> compare(fh, sh, comparator) <= 0
                ? first.left().flatMap(fl -> first.right().map(fr ->
                    heap(fh, fl, merge(fr, second, comparator)))
                : second.left().flatMap(sl -> second.right().map(sr ->
                    heap(sh, sl, merge(first, sr, comparator))))))
            .getOrElse(first.isEmpty()
                ? second
                : first);
    }

```

第二个方法使用一个被称为 `compare` 的辅助方法：

```

@SuppressWarnings("unchecked")
public static <A> int compare(A first, A second,
                             Result<Comparator<A>> comparator) {
    return comparator.map(comp -> comp.compare(first, second))
        .getOrElse(() -> ((Comparable<A>) first).compareTo(second));
}

```

这个方法强制转换了它的一个参数,但是你知道并没有抛出 `ClassCastException` 的风险,因为你确定如果类型参数没有继承 `Comparable`,堆是不可能在没有比较器的情况下被创建的。

现在可以删除 `static`、`final` 的 `EMPTY` 单例了。`add` 方法也需要进行如下修改：

```

public Heap<A> add(A element) {
    return merge(this, heap(element, this.comparator()));
}

```

最后, `Empty` 类中的 `left` 和 `right` 方法需要进行如下修改：

```

public Result<Heap<A>> left() {
    return Result.success(empty(this.comparator));
}

protected Result<Heap<A>> right() {
    return Result.success(empty(this.comparator));
}

```

练习 11.9

到目前为止,你只能通过 `merge` 方法将一个元素添加到一个 `Heap` 中。实现一

个 insert 方法，它不需要 merge 就可以添加一个元素。在 Heap 父类中定义一个抽象方法，其签名如下：

```
public abstract Heap<A> insert(A a)
```

提示

你应该重用练习 11.8 中的 compare 方法。

答案 11.9

Empty 的实现只是调用 heap 工厂方法，传入待插入的值和两个 this 的引用：

```
public Heap<A> insert(A a) {
    return heap(a, this, this);
}
```

在 H 类中，你要实现的算法很简单。让我们把待插入的元素称为 a。你需要用一个 head、一个 left 和一个 right 来构建一个新的 H：

- 如果这个 head 小于 a，请保持当前的 head。否则使用 a。
- 保持左分支原封不动。
- 如果 head 大于 a，则将 head 递归插入右分支。
- 否则，将 a 递归地插入右分支。

代码如下所示：

```
public Heap<A> insert(A a) {
    return heap(compare(head, a, comparator) < 0
        ? head
        : a, left, right.insert(compare(head, a, comparator) > 0
            ? head
            : a));
}
```

这段代码并没有被优化，因为你用相同的参数调用了两次 compare。你可以调用它一次并缓存结果，这也可使代码更容易阅读：

```
public Heap<A> insert(A a) {
    int comp = compare(head, a, comparator);
    return heap(comp < 0
        ? head
        : a, left, right.insert(comp > 0 ? head : a));
}
```

看起来还行？并非如此。

练习 11.10

练习 11.9 的答案运行在一个 `Heap<Integer>` 上可以工作，但是它有一个 bug。找到并将其修复。当然，如果你做过了练习 11.9，并直接发现了正确的解决方案，那就可以休息一下了。

提示

思考当插入的值与 `head` 的优先级相同时会发生什么。

答案 11.10

如果 `head` 的优先级等于插入元素 `a` 的优先级，则将 `a` 用于新 `head`，然后插入新的右分支。对于整型堆而言，这并没有什么大不了的，但是对于大多数其他类型而言可能就是一个大 bug 了。思考以下类型：

```
class Point implements Comparable<Point> {

    public final int x;
    public final int y;

    private Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public String toString() {
        return "(" + x + ", " + y + ")";
    }

    @Override
    public int compareTo(Point that) {
        return this.x < that.x ? -1 : this.x > that.x ? 1 : 0;
    }
}
```

这个类型表示可以仅比较其 `x` 坐标的点。现在，思考这段程序来模拟往堆中插入点：

```
List<Tuple<Integer, Integer>> points =
    List.list(1, 2, 2, 2, 6, 7, 5, 0, 5, 1).zipWithPosition();
Heap<Point> heap = points.foldLeft(Heap.empty(), h -> t ->
    h.insert(new Point(t._1, t._2)));
List<Point> lp = List.unfold(heap, hp -> hp.head())
```



```

        .flatMap(h -> hp.tail().map(t -> new Tuple<>(h, t))));
System.out.println(points);
System.out.println(lp);

```

插入点后，它们会按优先级顺序再次提取到列表中。结果如下（第一行显示原来的点）：

```

[(1,0), (2,1), (2,2), (2,3), (6,4), (7,5), (5,6), (0,7), (5,8), (1,9), NIL]
[(0,7), (1,9), (1,9), (2,3), (2,1), (2,3), (5,8), (5,6), (6,4), (7,5), NIL]

```

在第二行中，可以看到你得到了两个 $x=1$ 的点，但得到的不是 $(1,0)$ 和 $(1,9)$ ，而是两个 $(1,9)$ 。对于 $x=2$ 的点来说也有同样的问题。如果你只是将整型插入堆中，这个问题便不会显露出来。

这才是正确的实现：

```

public Heap<A> insert(A a) {
    int comp = compare(head, a, comparator);
    return heap(comp < 0
        ? head
        : a, left, right.insert(comp >= 0
            ? head
            : a));
}

```

经过这个小改动，结果更加正确了

现在你将会得到以下的（正确）结果：

```

[(1,0), (2,1), (2,2), (2,3), (6,4), (7,5), (5,6), (0,7), (5,8), (1,9), NIL]
[(0,7), (1,9), (1,0), (2,3), (2,1), (2,2), (5,8), (5,6), (6,4), (7,5), NIL]

```

11.5 总结

- 树可以被平衡，以获得更好的性能并避免在递归操作中溢出栈。
- 红黑树是一个自平衡的树结构，可以让你免于关注树的平衡。
- 可以通过委托到存储键 / 值元组的树来实现 map。
- 键不可比较的 map 需要处理冲突，以便存储那些表示相同键的元素。
- 优先队列允许以优先顺序获取元素的结构。
- 优先队列可以使用左倾堆来实现，左倾堆是一棵堆有序的二叉树。
- 可以使用额外的比较器来构建不可比较元素的优先队列。

用函数式的方式 处理状态改变

本章要点

- 创建一个函数式随机数发生器
- 设计一个通用 API 用于处理状态改变
- 处理和复合状态操作
- 使用递归状态操作
- 通用状态处理
- 构建一个状态机

在本章中，你将学习如何以纯函数式的方式处理状态。在前几章中尽量避免了改变状态，也许会让你感觉改变状态与函数式编程不兼容。并非如此。在函数式编程中，完全可以完美地处理状态改变。与你可能习惯的唯一区别是必须函数式地处理状态改变，即不借助副作用。

对于程序员来说，有很多处理状态改变的理由。随机数发生器是最简单的例子之一。随机数发生器是一个有方法能返回随机数的组件。如果随机数发生器没有状态（这意味着实际上没有改变状态），它将始终返回相同的数字。而这并不是你所期望的。

另一方面，因为在前面的章节中我已经说过很多次了，给定相同参数的函数应

该返回相同的值，所以可能难以想象这样的一个发生器如何能够工作。

12.1 一个函数式的随机数发生器

随机数发生器有很多用途，主要分为两大类：

- 在给定范围内生成均匀分布的数字。
- 生成真正的“随机”数字，即你无法预测的数字。

在第一种情况下，你不需要真正随机的数字，你需要的是随机分布。因而在这种情况下，随机性不适用于单个数字，而适用于一组数字。不仅如此，你会希望在需要时重新生成这组数字。这将允许你测试程序。如果生成的数字真的是随机的（在不可预测的角度上），那你将无法测试使用它的发生器或程序，因为你不知道期望的值。

在第二种情况下，你真的希望数字是不可预测的。例如，如果想要生成随机测试数据来测试程序，那么每当运行测试时生成相同的数据便徒劳无功了。

Java 有一个随机数发生器。你可以通过调用 `nextInt` 等方法来使用：

```
Random rng = new Random();
System.out.println(rng.nextInt());
System.out.println(rng.nextInt());
System.out.println(rng.nextInt());
```

这个程序打印出……好吧，你不知道。在每次运行时，它将打印不同的结果，如下所示：

```
773282358
-496891854
-47242220
```

虽然这有时是你想要的，但这不是函数式的。随机数发生器的 `nextInt` 方法不是一个函数，因为当使用相同的参数调用时，它并不总是返回相同的值。

没有参数的函数 `nextInt` 没有参数其实无关紧要。要成为一个函数，它必须总是返回相同的值。实际上，没有参数意味着它可以接收任何参数，而这个参数并不影响返回值。这与函数式的定义没有矛盾。这种函数只是一个简单的常数而已。

我们来看看发生了什么事情。如果该方法没有参数并返回一个值，则这个值必须来自什么地方。当然，你会猜到这个地方在随机数发生器内。每个调用都会改变值其实意味着在每个调用之间发生器都变化了；它有一个可变状态。所以问题是

`nextInt` 方法的返回值是否仅依赖于发生器的状态，还是依赖于别的什么。

如果返回的值仅仅取决于发生器的状态，则很容易便能使其成为函数式。你只需将发生器的状态作为参数传递给该方法即可。当然，由于方法返回的结果将会导致状态变化（为了让发生器不总是返回相同的值），该方法将返回生成的值和发生器的状态。你知道如何简单地通过返回一个元组来处理，所以 `nextInt` 方法的签名将进行如下改变：

```
public Tuple<Integer, Random> nextInt(Random)
```

这里的问题是，Java 的 `Random` 发生器不能以这种方式工作。`nextInt` 方法的返回值不仅取决于发生器的状态，还取决于系统时钟：系统时钟用于初始化发生器。事实上，Java 的 `Random` 发生器需要一个 `long` 值来初始化自己。从这一点上看，生成的一组数字将不会变化，但是这个被称为种子（seed）的 `long` 值，默认取决于系统时钟返回的纳秒数。（更多详细信息，请查看 `Random.java` 的源代码。）重要的是，Java 采用的方式是返回不可预测的数字，除非提供特定的种子来初始化发生器。所以你仍然可以用它以函数式的方式生成随机数字。

12.1.1 随机数发生器接口

你现在将实现一个函数式的随机数发生器。这并不是随机数发生器的最佳示例，但是由于你只是学习如何以函数式的方式处理状态改变，因此它将成为函数式状态处理的示例。

首先，你需要定义发生器的接口。生成随机数可以通过许多不同的方式完成，因此你可以使用不同的实现。从商业角度上看，发生器的质量取决于通过查看上一个数字来预测下一个数字的不可能程度。所以你可以定义一个简单的发生器，以低成本生成某种程度上可预测的数据，或者定义一个复杂的实现，来作为不可预测性对于安全而言至关重要的用例。

你的发生器接口如下：

```
import com.fpinjava.common.Tuple;

public interface RNG {
    Tuple<Integer, RNG> nextInt();
}
```

12.1.2 实现随机数发生器

在本节中，你将用 Java 的 Random 类的算法的简化版来尽量简单地实现随机数发生器：

```
import com.fpinjava.common.Tuple;

public class JavaRNG implements RNG {

    private final long seed;

    private JavaRNG(long seed) {
        this.seed = seed;
    }

    private JavaRNG() {
        this(System.currentTimeMillis());
    }

    private long nextSeed(long seed) {
        return (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
    }

    @Override
    public Tuple nextInt() {
        return new Tuple<>((int)(seed >>> 16), new JavaRNG(nextSeed(seed)));
    }

    public static RNG rng(long seed) {
        return new JavaRNG(seed ^ 0x5DEECE66DL & ((1L << 48) - 1));
    }

    public static RNG rng() {
        return new JavaRNG();
    }
}
```

所有要做的就是创建一个直接调用的组件，使随机数发生器更加函数式：

```
import com.fpinjava.common.Tuple;

public class Generator {
    public static Tuple<Integer, RNG> integer(RNG rng) {
        return rng.nextInt();
    }
}
```

为了了解如何使用这个类，让我们来看一个单元测试：

```
public void testInteger() {
    RNG rng = JavaRNG.rng(0);
    Tuple<Integer, RNG> t1 = Generator.integer(rng);
    assertEquals(Integer.valueOf(384748), t1._1);
    Tuple<Integer, RNG> t2 = Generator.integer(t1._2);
    assertEquals(Integer.valueOf(-1155484576), t2._1);
    Tuple<Integer, RNG> t3 = Generator.integer(t2._2);
    assertEquals(Integer.valueOf(-723955400), t3._1);
}
```

你可以看到，Generator 类的 integer 方法是函数式的。你可以按需多次运行这个测试，它会始终生成相同的值。发生器与 java.util.Random 不同，没有可变状态，并且 integer 方法是引用透明的。

如果你需要生成真正不可预测的数字，可以使用一个“随机”的长整型来调用 JavaRNG.rng 方法。例如，可以用 System.nanoTime() 的返回值。但请注意，返回值的精度到不了 1 纳秒，因此几个连续的调用可能会返回相同的值。这可以通过这种方法来避免：缓存 nanoTime 的返回值，如果值没有改变就再次调用，直到获得不同的值为止。java.util.Random 类提供了这样的服务，不过这里的代码已经简化过了。不过再说一遍，本章讨论的不是关于编写一个很棒的发生器，而是关于函数的状态处理。

练习 12.1

在 Generator 类中编写一个方法以返回一个小于参数但是大于等于 0 的随机正整数。签名如下：

```
public static Tuple <Integer, RNG> integer (RNG rng, int limit)
```

答案 12.1

只需从发生器中取得下一个随机值即可。用第一个元组成员除以参数的余数的绝对值来创建一个新的元组。第二个成员保持不变。

```
public static Tuple<Integer, RNG> integer(RNG rng, int limit) {
    Tuple<Integer, RNG> random = rng.nextInt();
    return new Tuple<>(Math.abs(random._1 % limit), random._2);
}
```

练习 12.2

编写一个方法以返回由 n 个随机整数组成的列表。它还需要返回转换为最新的

RNG 的当前状态，因而可以用于生成下一个整数。签名如下：

```
Tuple<List<Integer>, RNG> integers(RNG rng, int length)
```

提示

尽量不要使用显式递归。使用 List 类中的方法，首先创建与请求大小一样的列表并将其折叠。请注意，如果生成一个随机数列表，你可能会以相反的顺序返回（如果那样更简单的话）。但是你必须确保返回的发生器是最新的，即它必须是 nextInt 方法的最后一个返回值。

答案 12.2

思路是创建一个预期长度的列表，然后用正确的函数折叠它。你将使用整型列表来执行这个操作：

```
List.range(0, length).foldLeft(identity, f);
```

这是替代命令式编程中的索引循环的一种常见模式。f 函数在这里忽略了列表中的整型。由空列表开始，这个函数将发生器生成的值添加到列表中。所以函数的类型似乎应该如下所示：

```
Function<List<Tuple<Integer, RNG>>, Function<Integer,  
List<Tuple<Integer, RNG>>>
```

但是这么做会带来一个问题。你很容易就能将生成的 List<Tuple<Integer, RNG>> 转换为 List<Integer>，但是为了重新构造一个 Tuple<List<Integer>, RNG>，需要得到列表中的最后一个 RNG。这是因为将列表折叠到另一个列表中会反转元素的顺序。其实随机值的顺序相反并没有什么影响，但是你需要访问最后返回的 RNG，由于折叠，它将出现在末尾的位置上。你必须反转列表才能访问到它，可是这样做既不高效又不巧妙。

一个更好的方案是在折叠整型列表时传递当前的 RNG。结果将是一个 Tuple<List<Tuple<Integer, RNG>>, RNG>，用于折叠的函数如下所示：

```
Function<Tuple<List<Tuple<Integer, RNG>>, RNG>, Function<Integer,  
Tuple<List<Tuple<Integer, RNG>>, RNG>>> f = tuple -> i -> {  
    Tuple<Integer, RNG> t = integer(tuple._2);  
    return new Tuple<>(tuple._1.cons(t), t._2);  
};
```

这个类型也许看上去令人生畏，但是即便如此，你也不应该显式地定义它。编译器能够推断出这种类型，所以你不必刻意去写。完整的折叠如下：

```
Tuple<List<Tuple<Integer, RNG>>, RNG> result = List.range(0, length)
    .foldLeft(new Tuple<>(List.list(), rng), tuple -> i -> {
    Tuple<Integer, RNG> t = integer(tuple._2);
    return new Tuple<>(tuple._1.cons(t), t._2);
});
```

现在你得到了一个 `Tuple<List<Tuple<Integer, RNG>>, RNG>`，很容易就能构造预期的结果：

```
public static Tuple<List<Integer>, RNG> integers(RNG rng, int length) {
    Tuple<List<Tuple<Integer, RNG>>, RNG> result = List.range(0, length)
        .foldLeft(new Tuple<>(List.list(), rng), tuple -> i -> {
        Tuple<Integer, RNG> t = integer(tuple._2);
        return new Tuple<>(tuple._1.cons(t), t._2);
    });
    List<Integer> list = result._1.map(x -> x._1);
    return new Tuple<>(list, result._2);
}
```

如你所见，由于构造单链表的方式，随机数的结果列表仍然顺序相反，但是你不必反转列表。你并不在乎首先生成的数字最后才出现。唯一重要的事情是，返回的 RNG 将生成正确的数字。

如果你愿意，可以如下实现这个方法：

```
public static Tuple<List<Integer>, RNG> integers2(RNG rng, int length) {
    List<Tuple<Integer, RNG>> result = List.range(0, length).
        foldLeft(List.list(), lst -> i -> lst.cons(integer(rng)));
    List<Integer> list = result.map(x -> x._1);
    Result<Tuple<List<Integer>, RNG>> result2 =
    result.headOption().map(tr -> new Tuple<>(list, tr._2));
    return result2.getOrElse(new Tuple<>(List.list(), rng));
}
```

在正常情况下 (`length > 0`)，
`tr._2` 就是你返回的 RNG

在此对应于 `length == 0` 的情况，返回 rng
和一个作为默认值的空列表

或者，你也可以使用显式递归：

```
public static Tuple<List<Integer>, RNG> integers3(RNG rng, int length) {
    return integers3_(rng, length, List.list()).eval();
}
```



```
private static TailCall<Tuple<List<Integer>, RNG>> integers3_(RNG rng,
    int length, List<Integer> xs) {
    if (length <= 0)
        return TailCall.ret(new Tuple<>(xs, rng));
    else {
        Tuple<Integer, RNG> t1 = rng.nextInt();
        return TailCall.sus(() ->
            integers3_(t1._2, length - 1, xs.cons(t1._1)));
    }
}
```

但是请注意，函数式程序员通常认为使用显式递归是不好的实践。他们更喜欢使用折叠来抽象递归。

12.2 处理状态的通用API

正如我所说，实现 RNG 的方式并不是实现发生器的最佳方式。这只是一个例子，向你展示如何以函数式的方式处理状态。你可以从这个例子中学到用 RNG 来表示发生器的当前状态。

但如果要生成整数，你可能对 RNG 毫无兴趣，可能更倾向于使其透明。换句话说，到现在为止，你使用的是一个接收 RNG 并返回生成值的函数，无论是 Integer、List 还是其他，还有新的 RNG：

```
Function<RNG, Tuple<A, RNG>>
```

如果你能摆脱 RNG，会不会更好？是否可以抽象 RNG 处理，以使你不必顾忌它呢？

要抽象 RNG 处理，你需要创建一个封装 RNG 参数的新类型：

```
public interface Random<A> extends Function<RNG, Tuple<A, RNG>>
```

你现在可以根据这种新类型重新定义生成操作。例如，你可以把如下方法：

```
public static Tuple<Integer, RNG> integer(RNG rng) {
    return rng.nextInt();
}
```

替换为如下函数：

```
public static Random<Integer> integer = RNG::nextInt;
```

12.2.1 使用状态操作

抽象出 RNG 后，所剩的内容与前几章中学过的参数化类型非常相似。你在此得到的是一些简单类型的计算上下文（**computational context**）。还记得 List 和 Result 吗？那些类型就像其他类型的计算上下文一样。

整型列表是 Integer 类型的计算上下文。例如，它允许你将整型列表应用于从 Integer 到另一种类型的函数，而无须关心列表中的元素数量。

Result 并无不同。它创建一个值的计算上下文，允许你将一个函数应用于该值，而无须关心值是否真实存在。以同样的方式，Random 允许你将计算应用于值，而无须处理该值为随机数的事实。

能用你为 List 和 Result 定义的相同抽象来定义 Random 吗？让我们来试一试。

首先，你需要一个办法通过一个值来创建 Random。虽然这在现实生活中似乎没有什么用，但还是需要用它来创建其他抽象。这个方法称为 unit：

```
public static <A> Random<A> unit(A a) {
    return rng -> new Tuple<>(a, rng);
}
```

unit 这个名称是约定俗成的。你也可以将它用于 Result、Stream、List、Heap 等，但是你选择了与业务更加相关的名称，例如 list 和 success。它是应用于不同类型的相同概念。

让我们来进一步尝试。你可以使用 A 到 B 的函数把 Random<A> 转换为 Random 吗？当然可以。对于其他类型，这被称为 map。让我们为 Random 定义一个 map 方法：

```
static <A, B> Random<B> map(Random<A> s, Function<A, B> f) {
    return rng -> {
        Tuple<A, RNG> t = s.apply(rng);
        return new Tuple<>(f.apply(t._1), t._2);
    };
}
```

这个方法可以定义在任何地方，例如在 Random 接口中。

练习 12.3

使用 map 方法生成随机 Boolean 值。通过在 Random 接口中创建一个函数来达成。

提示

用你刚刚创建的以下函数：

```
Random<Integer> intRnd = RNG::nextInt;
```

答案 12.3

该答案包括了将 `intRnd` 函数返回的结果映射到把 `int` 转换为 `boolean` 的函数。当然，如果希望结果为 `true` 的概率有 50%，你需要选择相应的函数。常用的算法是检查除以 2 的余数是否为 0：

```
Random<Boolean> booleanRnd = Random.map(intRnd, x -> x % 2 == 0);
```

练习 12.4

实现返回随机生成 `Double` 的函数。

答案 12.4

这与 `booleanRnd` 函数完全相同。唯一的区别是要映射的函数：

```
Random<Double> doubleRnd =  
  map(intRnd, x -> x / (((double) Integer.MAX_VALUE) + 1.0));
```

12.2.2 复合状态操作

在上一节中，你复合了普通函数与状态操作。如果你需要复合两个或更多的状态操作怎么办？这正是你在练习 12.2 中所做的生成随机整型列表。你可以在 `Random` 类型中抽象它吗？一开始，你可能需要复合两个 `Random` 实例的方法，例如生成一对随机数。

练习 12.5

实现一个接收 `RNG` 并返回一对整数的函数。

提示

首先在 `Random` 接口中定义一个 `map2` 方法，用于复合对随机数发生器的两次调用，以生成一对泛型 `A` 和 `B` 的值，然后把它们作为返回第三种类型 `C` 的函数参数。它的签名如下：

```
static <A, B, C> Random<C> map2(Random<A> ra, Random<B> rb,
                                Function<A, Function<B, C>> f) {
```

答案 12.5

这不会比实现 `map` 还难。首先你需要将 `rng` 参数传递给第一个函数。然后，从结果中获得返回的 RNG，并将其传给第二个函数。最后，使用两个值作为 `f` 函数的输入，并将结果与第二次生成的 RNG 一起返回：

```
static <A, B, C> Random<C> map2(Random<A> ra, Random<B> rb,
                                Function<A, Function<B, C>> f) {
    return rng -> {
        Tuple<A, RNG> t1 = ra.apply(rng);
        Tuple<B, RNG> t2 = rb.apply(t1._2);
        return new Tuple<>(f.apply(t1._1).apply(t2._1), t2._2);
    };
}
```

用这个方法，你可以定义返回一对随机整数的函数，如以下示例所示：

```
Random<Tuple<Integer, Integer>> intPairRnd =
    map2(intRnd, intRnd, x -> y -> new Tuple<>(x, y));
```

不要对这两个值使用相同的 RNG，这样做会生成一对相同的整数！

练习 12.6

实现一个函数，接收 RNG 并返回随机生成的整型列表。

提示

整个过程很容易描述。首先，你需要生成一个 `List<Random<Integers>>`。然后，需要将其转换为 `Random<List<Integer>>`。这是否提醒了你什么？你对 `Result` 实现了同样的抽象，把 `List<Result>` 更改为 `Result<List>`，并称它为 `sequence`。

你可以开始从 `Random` 类中实现 `sequence` 方法。这是它的签名：

```
static <A> Random<List<A>> sequence(List<Random<A>> rs)
```

你可以使用 `List` 类中定义的 `List.fill()` 方法来生成列表，它的签名如下：

```
public static <A> List<A> fill(int n, Supplier<A> s)
```

答案 12.6

你应该能猜到必须遍历列表。无须使用显式递归，而且你也不应该这么做！应该用折叠来代替。起始值是一个用空列表构造的 `Random`。从此 `unit` 方法开始成为一个有用的工具。对 `foldLeft` 或 `foldRight` 使用一个将 `map2` 应用于当前累加器的值和待处理的列表元素的函数。

写描述比写代码要困难得多。以下是使用了 `foldLeft` 的示例：

```
static <A> Random<List<A>> sequence(List<Random<A>> rs) {
    return rs.foldLeft(unit(List.list()), acc -> r ->
        map2(r, acc, x -> y -> y.cons(x)));
}
```

然后定义返回随机整型列表的函数。此时的类型不再是 `Random<Integer>` 了，因为你需要处理代表列表所需长度的附加 `int` 参数：

```
Function<Integer, Random<List<Integer>>> integersRnd =
    length -> sequence(List.fill(length, () -> intRnd));
```

把这个实现与练习 12.2 的答案进行比较，就会发现有点意思：

```
public static Tuple<List<Integer>, RNG> integers(RNG rng, int length) {
    Tuple<List<Tuple<Integer, RNG>>, RNG> result = List.range(0, length)
        .foldLeft(new Tuple<>(List.list(), rng), tuple -> i -> {
        Tuple<Integer, RNG> t = integer(tuple._2);
        return new Tuple<>(tuple._1.cons(t), t._2);
    });
    List<Integer> list = result._1.map(x -> x._1);
    return new Tuple<>(list, result._2);
}
```

你可以看到折叠已被抽象为 `sequence` 方法，而中间结果处理也已被抽象为 `map2` 方法。编码的结果非常整洁并容易理解（只要你理解这两个抽象）。在 `integersRnd` 函数中，你无须操作 `RNG` 发生器。`sequence` 和 `map2` 方法亦是如此。如你所见，离实现一个通用的状态处理工具已经非常接近了。

12.2.3 递归状态操作

到目前为止，你已经见过了如何多次调用发生器以返回多个值。但是你可能需要处理不同的用例。假设你打算生成不是 5 的倍数的整数。

如果你在编写一个命令式程序，可以简单地生成一个数字并进行检查。如果不

是 5 的倍数，将其返回，否则生成下一个数字。在这个实现中，平均 5 次就有一次需要生成第二个数字。你可能会想要这样做：

```
Random<Integer> notMultipleOfFiveRnd = Random.map(intRnd, x -> {
  return x % 5 != 0
    ? x
    : Random.notMultipleOfFiveRnd.apply(???);
});
```

但是，如何访问必须传给递归调用 notMultipleOfFiveRnd 函数的 RNG？这是第一次函数调用所生成的 RNG。

你可以通过显式处理第一次函数调用的结果来解决这个问题：

```
Random<Integer> notMultipleOfFiveRnd = rng -> {
  Tuple<Integer, RNG> t = intRnd.apply(rng);
  return t._1 % 5 != 0
    ? t
    : Random.notMultipleOfFiveRnd.apply(t._2);
};
```

但你似乎回到了起点。你真正需要的是一个 flatMap 方法。

练习 12.7

编写一个 flatMap 方法并用它实现 notMultipleOfFiveRnd 函数。以下是 flatMap 方法的签名：

```
static <A, B> Random<B> flatMap(Random<A> s, Function<A, Random<B>> f)
```

答案 12.7

flatMap 方法与 map 方法非常相似：

```
static <A, B> Random<B> flatMap(Random<A> s, Function<A, Random<B>> f) {
  return rng -> {
    Tuple<A, RNG> t = s.apply(rng);
    return f.apply(t._1).apply(t._2);
  };
}
```

不同之处在于，你只需把生成的值传给 f 函数，它便返回一个 Random，而不必构造一个元组并将其返回。要记住这实际上是一个 Function<RNG, Tuple<A, RNG>>，所以你把应用 s 得到的 RNG 传给该函数，它便返回一个可以返回的 Tuple<A, RNG>。

现在你可以通过 flatMap 来实现 notMultipleOfFiveRnd 函数：

```
Random<Integer> notMultipleOfFiveRnd = Random.flatMap(intRnd, x -> {
    int mod = x % 5;
    return mod != 0
        ? unit(x)
        : Random.notMultipleOfFiveRnd;
});
```

练习 12.8

通过 flatMap 实现 map 和 map2。

提示

map、flatMap 和 unit 之间有一个关系：flatMap 是 map 和 unit 的组合。

答案 12.8

以下是两个新的实现：

```
static <A, B> Random<B> map(Random<A> s, Function<A, B> f) {
    return flatMap(s, a -> unit(f.apply(a)));
}

static <A, B, C> Random<C> map2(Random<A> ra, Random<B> rb,
                                Function<A, Function<B, C>> f) {
    return flatMap(ra, a -> map(rb, b -> f.apply(a).apply(b)));
}
```

如你所见，flatMap 给了你一个额外的抽象级别，它允许你编写更清晰的方法实现。

12.3 通用状态处理

到目前为止，本章开发的所有方法和函数都曾用于生成随机数。不过你一开始的代码是专门用于生成随机数的，但是最后做出来的工具却与生成随机数完全无关。Random 接口的方法与生成随机数相关仅仅是因为这个接口事实上继承了 Function<RNG, Tuple<A, RNG>>。其实你可以重新定义这个接口来处理任何状态：

```
interface State<S, A> extends Function<S, Tuple<A, S>> {}
```

你当然知道组合比继承更好，因此你可能更愿意用一个委托来定义 State 类：

```
public class State<S, A> {
    public final Function<S, Tuple<A, S>> run;

    public State(Function<S, Tuple<A, S>> run) {
        super();
        this.run = run;
    }
}
```

现在你可以把 Random 重新定义为 State 的一个特殊情况：

```
public class Random<A> extends State<RNG, A> {
    public Random(Function<RNG, Tuple<A, RNG>> run) {
        super(run);
    }
}
```

练习 12.9

通过以通用的方式重新实现 Random 接口的方法来完成 State 类。

提示

将方法定义为实例方法，当然除了需要为静态的 unit 方法以外，需要为每个方法都创建一个新的 State。

答案 12.9

你的新方法如下：

```
public static <S, A> State<S, A> unit(A a) {
    return new State<>((state) -> new Tuple<>(a, state));
}

public <B> State<S, B> map(Function<A, B> f) {
    return flatMap(a -> State.unit(f.apply(a)));
}

public <B, C> State<S, C> map2(State<S, B> sb, Function<A,
    Function<B, C>> f) {
    return flatMap(a -> sb.map(b -> f.apply(a).apply(b)));
}

public <B> State<S, B> flatMap(Function<A, State<S, B>> f) {
```



```

return new State<> (s -> {
    Tuple<A, S> temp = run.apply(s);
    return f.apply(temp._1).run.apply(temp._2);
});
}

public static <S, A> State<S, List<A>> sequence(List<State<S, A>> fs) {
    return fs.foldRight(State.unit(List.<A>list()),
        f -> acc -> f.map2(acc, a -> b -> b.cons(a)));
}

```

现在，你可以用 `State<RNG, A>` 的别名来替换 `Random` 接口：

```

public class Random<A> extends State<RNG, A> {
    public Random(Function<RNG, Tuple<A, RNG>> run) {
        super(run);
    }
    public static State<RNG, Integer> intRnd = new Random<> (RNG::nextInt);
}

```

12.3.1 状态模式

假如你需要生成三个随机整数来初始化一个三维的（3D）点：

```

public class Point {

    public final int x;
    public final int y;
    public final int z;
    public Point(int x, int y, int z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    @Override
    public String toString() {
        return String.format("Point(%s, %s, %s)", x, y, z);
    }
}

```

你可以如下创建一个随机的 `Point`：

```

State<RNG, Point> ns =
    intRnd.flatMap(x ->
        intRnd.flatMap(y ->
            intRnd.map(z -> new Point(x, y, z))));

```

这段代码只修改一个状态。但是如果你有一个用于读取状态的 `get` 方法和一个

用于写入的 `set` 方法，则可以简化这段代码。你可以通过用函数 `f` 复合它们来修改状态，如下所示：

```
public static <S> State<S, Nothing> modify(Function<S, S> f) {
    return State.<S>get().flatMap(s -> set(f.apply(s)));
}
```

该方法返回一个 `State<S, Nothing>`，因为它并没有返回一个值。你只对修改状态感兴趣。需要如下定义一个 `Nothing` 类型：

```
public final class Nothing {

    private Nothing() {}

    public static final Nothing instance = new Nothing();
}
```

不用 `Nothing` 类型的话，你可以返回 `Void`，但是实例化 `Void` 有一些棘手，得用点肮脏的手段，所以我们更倾向于整洁的方案。

`get` 方法创建了一个函数，它简单地将参数的状态作为状态和值返回：

```
public static <S> State<S, S> get() {
    return new State<>(s -> new Tuple<>(s, s));
}
```

`set` 方法创建一个函数，它返回的新状态为参数的状态，值为 `Nothing` 单例：

```
public static <S> State<S, Nothing> set(S s) {
    return new State<>(x -> new Tuple<>(Nothing.instance, s));
}
```

12.3.2 构建一个状态机

状态机是复合状态变化的常用工具之一。状态机是一段代码，它通过有条件地从一个状态切换至另一个状态来处理输入。许多业务问题可以表示为条件状态变化。

通过创建参数化状态机，你可以抽象出有关状态处理的所有细节。这样，就可以通过简单地列出条件 / 转换对来处理任何这类问题，然后提供输入列表以取得结果状态。状态机将透明地处理各种转换的组合。

首先，你要定义两个接口以表示条件和相应的转换。这些接口不是绝对必需的，因为它们只是简单的函数，不过可以简化代码：

```
interface Condition<I, S> extends Function<StateTuple<I, S>, Boolean> {}
interface Transition<A, S> extends Function<StateTuple<A, S>, S> {}
```

StateTuple 类也是一个用于简化代码的辅助类。它就是一个元组，两个字段被称为 value 和 state。这样比 _1 和 _2 或者 left 和 right 更加易读，因为很容易就会忘了哪个是哪个。

```
public class StateTuple<A, S> {

    public final A value;
    public final S state;

    public StateTuple(A a, S s) {
        value = a;
        state = s;
    }
}
```

StateMachine 类仅是持有一个类型为 Function<A, State<S, Nothing>> 的函数。一个选择是将最终值作为状态的一部分返回。此处的最终值包含在状态中，所以你无须单独处理该值。

状态机由 <Tuple<Condition<A, S>, Transition<A, S>> 的列表构成。在构造函数中，函数如下构建：

```
public class StateMachine<A, S> {

    Function<A, State<S, Nothing>> function;

    public StateMachine(List<Tuple<Condition<A, S>,
                                Transition<A, S>>> transitions) {
        function = a -> State.sequence(m ->
            Result.success(new StateTuple<>(a, m)).flatMap((StateTuple<A, S> t) ->
                transitions.filter((Tuple<Condition<A, S>, Transition<A, S>> x) ->
                    x._1.apply(t)).headOption().map((Tuple<Condition<A, S>,
                        Transition<A, S>> y) -> y._2.apply(t))).getOrElse(m));
    }
}
```

State.sequence 方法的定义如下所示：

```
public static <S> State<S, Nothing> sequence(Function<S, S> f) {
    return new State<>(s -> new StateTuple<>(Nothing.instance, f.apply(s)));
}
```

这段代码似乎很复杂，但它仅仅是构建一个函数而已，这个函数将复合构造函数所接收的参数的所有条件转换。

StateMachine 类还定义了一个接收输入列表以生成结果状态的 process 方法：

```
public State<S, S> process(List<A> inputs) {
    List<State<S, Nothing>> a = inputs.map(function);
    State<S, List<Nothing>> b = State.compose(a);
    return b.flatMap(x -> State.get());
}
```

State.compose() 方法的定义如下所示：

```
public static <S, A> State<S, List<A>> compose(List<State<S, A>> fs) {
    return fs.foldRight(State.unit(List.<A>list()),
        f -> acc -> f.map2(acc, a -> b -> b.cons(a)));
}
```

练习 12.10

编写一个模拟自动柜员机的 Atm 类。输入由以下接口表示：

```
public interface Input {

    Type type();

    boolean isDeposit();

    boolean isWithdraw();

    int getAmount();

    enum Type {DEPOSIT, WITHDRAW}
}
```

Input 接口将有 Deposit 和 Withdraw 两个实现：

```
public class Deposit implements Input {

    private final int amount;

    public Deposit(int amount) {
        super();
        this.amount = amount;
    }

    @Override
    public Type type() {
        return Type.DEPOSIT;
    }
}
```

```

@Override
public boolean isDeposit() {
    return true;
}

@Override
public boolean isWithdraw() {
    return false;
}

@Override
public int getAmount() {
    return this.amount;
}
}

```

```

public class Withdraw implements Input {

```

```

    private final int amount;

```

```

    public Withdraw(int amount) {
        super();
        this.amount = amount;
    }

```

```

@Override
public Type type() {
    return Type.WITHDRAW;
}

```

```

@Override
public boolean isDeposit() {
    return false;
}

```

```

@Override
public boolean isWithdraw() {
    return true;
}

```

```

@Override
public int getAmount() {
    return this.amount;
}
}

```

为了简化代码，附加一个 Outcome 类以表示结果元组：

```

public class Outcome {

    public final Integer account;

```

```

public final List<Integer> operations;

public Outcome(Integer account, List<Integer> operations) {
    super();
    this.account = account;
    this.operations = operations;
}

public String toString() {
    return "(" + account.toString() + "," + operations.toString() + ")";
}
}

```

如你在此类中所见，Atm 会生成一个表示该账户余额的整型，以及表示操作金额的整型列表（存款为正数，取款为负数）。

这个练习就是实现 Atm 类，它基本上就包含一个构造 StateMachine 的方法：

```

public class Atm {
    public static StateMachine<Input, Outcome> createMachine() {
        ...
    }
}

```

提示

createMachine 的实现必须首先构造一个条件和相应转换的元组列表。这些元组需要被排序，越特定的越靠前。最后一个元组需要包罗万象的条件。这与 switch 结构中的 default 类似（还与练习 3.2 中的 DefaultCase 类似）。这种包罗万象的条件并不总是需要，但是有一个终归更安全。元组列表将被用作 StateMachine 构造函数的参数。

你需要运行状态机以获得可观测的结果。可以通过对起始状态应用 run 函数来搞定，它将生成一个结果状态，你可以从中获取值：

```

Outcome out = Atm.createMachine().process(inputs)
               .run.apply(new Outcome(0, List.list())) .value;

```

通过增加以下方法，可以把这段代码的运行部分（第二行）抽象到 State 类中：

```

public A eval(S s) {
    return run.apply(s).value;
}

```

通过这个增加的方法，运行状态机更加简洁：

```
Outcome out = Atm.createMachine().process(inputs)
    .eval(new Outcome(0, List.list()));
```

答案 12.10

答案就像一个命令式语言的程序。它可以用以下伪代码描述：

处理操作

如果是存款操作

把金额加入账户，并将操作添加到操作列表中

处理下一个操作

如果是取款操作，并且金额比账户余额要少

从账户中减去金额，并将操作添加到操作列表中

处理下一个操作

否则

不改变账户与操作列表

很容易就能实现：

```
public static StateMachine<Input, Outcome> createMachine() {

    Condition<Input, Outcome> predicate1 = t -> t.value.isDeposit();
    Transition<Input, Outcome> transition1 =
        t -> new Outcome(t.state.account + t.value.getAmount(),
            t.state.operations.cons(t.value.getAmount()));

    Condition<Input, Outcome> predicate2 = t -> t.value.isWithdraw()
        && t.state.account >= t.value.getAmount();
    Transition<Input, Outcome> transition2 =
        t -> new Outcome(t.state.account - t.value.getAmount(),
            t.state.operations.cons(- t.value.getAmount()));
    Condition<Input, Outcome> predicate3 = t -> true;
    Transition<Input, Outcome> transition3 = t -> t.state;

    List<Tuple<Condition<Input, Outcome>,
        Transition<Input, Outcome>>> transitions = List.list(
        new Tuple<>(predicate1, transition1),
        new Tuple<>(predicate2, transition2),
        new Tuple<>(predicate3, transition3));
    return new StateMachine<>(transitions);
}
```

如果你想看到机器上的实战，只需运行本书附带代码中的单元测试即可。

这段代码就像一个命令式程序般工作，顺便说一句，它就是。它是函数式地完成命令式程序。当然，使用这种代码来处理这样一个简单的问题有些杀鸡用牛刀了。这种方式的主要缺点不在于代码的复杂性（这段代码很简单），而是它太冗长了。另

一方面，它的优势是可以将其以近乎于零的成本进行扩展。你需要做的全部，就是在正确的地方插入正确的条件 / 转换。

练习 12.11

修改上一段程序，以报告诸如提取金额试图超过账户余额这样的错误。

答案 12.11

我没有这个练习的书面答案，但是我在本书附带的代码中提供了一个可行的方案以及相应的 JUnit 测试。

12.3.3 何时使用状态和状态机

函数式地处理状态似乎是一个过于复杂的命令式编程版本。对于可以在书中描述的简单且小巧的例子而言确实如此。但是，如果你考虑具有大量规则的复杂程序，则函数式状态处理的高级别抽象显然是有益的。但这并非是唯一优点——主要优点是可扩展性。你可以简单地通过修改规则或增加更多规则来演进应用程序，而无须冒着任何把实现搞砸的风险。

你还可以让它更简单。用 Java 描述规则（条件 / 转换）非常冗长，但是可以用更简洁的形式来编写它们，到那时你只需读懂并将它们转换为 Java。

这可能会演进成创建一个领域特定语言（DSL）。当然，你需要一个解析器来处理用这个 DSL 编写的程序，不过可以用函数式状态机轻松创建这样的解析器。（状态机并非解析所有语法类型的最佳解决方案，但那是另一回事了。）

12.4 总结

- 生成随机数涉及管理随机数发生器的状态。
- 通过对状态操作表示的使用，可以用函数式的方式管理状态。
- 可以借助 `map` 和 `flatMap` 等方法来复合状态操作。
- 可以递归地复合状态操作。
- `State` 类型是状态操作的通用表示形式，可以用作实现状态机的基础。

13

函数式输入/输出

本章要点

- 在上下文内安全地应用作用
- 为 `Result` 和 `List` 添加作用的应用
- 复合成功和失败的作用
- 通过 `Reader` 的抽象，安全地从控制台、文件或内存中读取数据
- 通过 `IO` 类型处理输入/输出

迄今为止，你已学会了如何编写任何有用的结果都没有真正生成的函数式程序。你学会了如何复合真正的函数来构建更强大的函数。更有意思的是，你学习了如何以安全、函数式的方式使用非函数式操作。非函数式操作是会生成副作用的操作，例如抛出异常、改变外界或是依赖外界来生成结果。例如，你学习了如何使用可能不安全的整数除法运算，并通过在计算上下文内使用它来将其转换为一个安全的运算。

你已经遇到过几个这样的计算上下文：

- 你在第 7 章中开发的 `Result` 类型正是这样的计算上下文，允许你使用能以

安全无误的方式生成错误的函数。

- 第 6 章中的 `Option` 类型也是一个计算上下文,用于安全地应用有时可能(对于一些参数而言)不生成数据的函数。
- 在第 5 章和第 8 章中学习的 `List` 类是一个计算上下文,它允许使用在元素集合的上下文中处理单个元素的函数,但并不处理错误。它还处理表示为空列表的缺失数据。

在学习这些类型和 `Stream`、`Map`、`Heap` 以及 `State` 等其他类型时,你并不关心生成有用的结果。然而在本章中,你将学习在函数式编程中生成有用结果的几种技术。它包括显示结果给人类用户或是将结果传递给另一个程序。

13.1 在上下文中应用作用

还记得你曾经将一个函数应用于整型运算的结果。假设你想编写一个 `inverse` 函数来计算一个整数的倒数:

```
Function<Integer, Result<Double>> inverse = x -> x != 0
    ? Result.success((double) 1 / x)
    : Result.failure("Division by 0");
```

该函数可以应用于一个整型值,但是当复合其他函数时,该值将会是另一个函数的输出,因此它一般已经在上下文中,并且常常是类型相同的上下文。以下是一个例子:

```
Result<Integer> ri = ...
Result<Double> rd = ri.flatMap(inverse);
```

值得注意的是,你不应该从 `ri` 上下文中把值取出来应用该函数。做法正好相反:将函数传递给上下文 (`Result` 类型),以使它可以在自己内部应用,生成一个新的上下文,可能还会把结果值包装起来。你在此将函数传递给 `ri` 上下文,生成新的结果 `rd`。

真是又简洁又安全。不会发生什么坏事,不会抛出任何异常。这就是函数式编程之美:无论用什么输入数据,你总是有一个可以工作的程序。但问题是,你要如何使用这个结果?假如你打算在控制台上显示结果——应该怎么做?

13.1.1 作用是什么

我将纯函数定义为没有任何可观测到的副作用的函数。作用是可以在程序外界观测到的东西。一个函数所扮演的角色就是返回一个值，而一个副作用就是除函数外界可观测到的返回值之外的任何东西。被称为副作用是因为它是返回值的附加部分。一个作用（没有“副”）就像是一个副作用，但它是一个程序扮演的主要（并且通常是唯一的）角色。函数式编程即关于以函数式的方式用纯函数（没有副作用）和纯作用来编写程序。

问题是，以函数式的方式处理作用到底是什么意思？目前我可以给出的最接近的定义是：“以不妨碍函数式编程原则的方式处理作用，其中最重要的原则是引用透明性。”有几种办法都可以达成这一目标，不过想要全部完成这个目标可能会很复杂。通常，接近它就足够了。你可以自行决定使用哪种技术。要使函数式程序生成可观测的作用，最简单（尽管不是完全函数式）的方式就是将作用应用于上下文。

13.1.2 实现作用

正如我刚才所说，作用是可以从程序外界观测到的东西。当然，为了有价值，这个作用一般都要反映出程序的结果，所以你通常要接收程序的结果并用它来做一些可观测的事。请注意，“可观测”并不总是意味着可以被人类操作者观测到。通常，另一个程序可以观测到结果，然后可能将该作用转换为可以被人类操作者观测到的东西，无论是同步还是异步。打印到计算机屏幕可以被操作者看到。另一方面，写入数据库可能并不总是对人类用户直接可见。有时人类会去查找结果，但它一般会被其他程序稍后读取。在第 14 章中，你将学习如何用这类作用与其他程序通信。

因为作用通常应用于一个值，所以可以把纯作用建模为一种不返回任何值的特殊函数。在本书中我通过以下接口来表示：

```
public interface Effect<T> {  
    void apply(T t);  
}
```

请注意，这相当于 Java 的 Consumer 接口，只是类和方法的名称不同。其实正如我在本书开头提到过几次的，名字没有关系，但有意义的名字更好。

Java 将 Effect 接口称为函数式接口，大意就是拥有单一抽象方法（single abstract method, SAM）的接口。要定义一个包括将 Double 值打印到屏幕的作用，

可以这样写：

```
Effect<Double> print = x -> System.out.println(x);
```

用方法引用或许更好：

```
Effect<Double> print = System.out::println;
```

请注意，这将创建一个类型为 `Effect<Double>` 的对象，因此它通常不是最有效的处理作用的办法。命名作用与命名函数类似：匿名 `lambda`（不要与匿名类混淆）一般编译为添加到底层代码的一些附加指令，却命名为“`lambda` 编译为对象”（`lambdas compile to objects`）。所以一般最好把作用用于匿名 `lambda` 或匿名方法引用。而且，使用匿名 `lambda` 可以让我们不必显式地声明这个类型。

下面这样就是你需要的，其中 `rd` 是 13.1 节中示例的 `Result`：

```
rd.map(x -> System.out.println(x));
```

不幸的是，这不能编译，因为表达式 `System.out.println(x)` 返回 `void`，而它必须返回一个值以使代码能够被编译。

你可以使用一个函数来返回值并作为副作用打印出来。只要忽略返回值即可。但你还可以做得更好，如第 7 章中所见。在那一章中，你在 `Result` 类中编写了一个 `forEach` 方法，该方法会接收作用并将其应用于所包装的值。这个方法在 `Empty` 类中的实现如下：

```
public void forEach(Effect<T> ef) {
    // Do nothing
}
```

在 `Success` 类中，它的实现就像这样：

```
public void forEach(Effect<T> ef) {
    ef.apply(value);
}
```

当然，你无法为这个方法编写单元测试。要验证它是否工作，可以运行清单 13.1 中显示的程序，并查看屏幕上的结果。

清单 13.1 输出数据

```
public class ResultTest {
    public static void main(String... args) {
```

```

Result<Integer> ra = Result.success(4);
Result<Integer> rb = Result.success(0);

Function<Integer, Result<Double>> inverse = x -> x != 0
    ? Result.success((double) 1 / x)
    : Result.failure("Division by 0");

Effect<Double> print = System.out::println;

Result<Double> rt1 = ra.flatMap(inverse);
Result<Double> rt2 = rb.flatMap(inverse);

System.out.print("Inverse of 4: ");
rt1.forEach(print);

System.out.print("Inverse of 0: ");
rt2.forEach(print);
}

```

模拟可能失败的函数返回的数据

输出结果值

因为没有值，所以不生成任何输出

该程序生成以下结果：

```

Inverse of 4: 0.25
Inverse of 0:

```

练习 13.1

在 List 类中编写一个 forEach 方法，该方法接收一个作用并将其应用于列表中的所有元素。

答案 13.1

Nil 类的实现与 Result.Empty 相同：

```

public void forEach(Effect<A> ef) {
    // Do nothing
}

```

Cons 类的最简单的递归实现如下：

```

public void forEach(Effect<A> ef) {
    ef.apply(head);
    tail.forEach(ef);
}

```

不幸的是，如果你有几千个元素，这个实现将会把栈击穿。

这个问题有很多不同的解决办法。你不能直接使用 TailCall 类来使递归变得

栈安全，但可以使用接收一个副作用的辅助函数并忽略结果：

```
public void forEach(Effect<A> ef) {
    forEach(this, ef).eval();
}

private static <A> TailCall<List<A>> forEach(List<A> list, Effect<A> ef) {
    return list.isEmpty()
        ? TailCall.ret(list)
        : TailCall.sus(() -> {
            ef.apply(list.head());
            return forEach(list.tail(), ef);
        });
}
```

这个实现用了一个 `forEach` 辅助函数的副作用，但是因为你实现的是应用一个作用，所以并不那么重要。另一个简单的（更有效率的）方案是使用一个 `while` 循环。你可以自己选择哪种实现。

13.1.3 用于失败情况的更强大的作用

虽然列表为空（对于 `Option.None` 和 `Result.Empty` 也是如此）时什么也不做合情合理，但是对于处理可能表示错误的结果，这当然还不够。在这种情况下，你可能需要对错误应用一个作用。

你的 `Result` 类会在出错时包含一个 `Exception`。你可能会想到这种情况有两种不同的作用。第一个作用是抛出异常，第二个是以其他方式处理异常以避免将其抛出。

在第 7 章中，你在 `Result` 类中编写了 `forEachOrThrow` 方法，它以 `Effect` 为参数，并将其应用于所含的值（如果存在的话），或者在 `Failure` 时抛出异常。

`forEachOrThrow` 的 `Empty` 实现什么事情也不做，与 `forEach` 的实现类似。`Failure` 的实现只是抛出包含的异常：

```
public void forEachOrThrow(Effect<T> c) {
    throw this.exception;
}
```

`Success` 的实现还是与 `forEach` 类似，并将作用应用于所含的值：

```
public void forEachOrThrow(Effect<T> e) {
    e.apply(this.value);
}
```

你通常不会想在失败时抛出异常，至少 `Result` 类是这样。一般来说，客户端决定要做什么，而你也许会希望做一些不像抛异常那样激进的事。例如，你可能需要在继续处理前先记录异常。

日志记录并不很函数式，因为日志一般都有副作用。没有程序会以记录日志为主要目标。用如 `forEach` 等方法来应用一个作用破坏了函数式契约。这本身不是一个问题，但是当你记录日志时，突然停止了函数式——这在一定程度上标志着函数式程序的结束。应用完作用之后，你又可以开始另一段新的函数式程序了。

如果你的应用程序对每个方法都记录日志，那么命令式和函数式编程之间的界限将会变得很模糊。但是由于记录日志通常是一个需求，至少在 Java 世界中是这样，你可能需要用一个整洁的方法来实现。在发生故障时，你没有简单记录异常的办法。你需要的是将一个 `failure` 转换为其异常的 `success`。为此，你需要直接访问异常，而这不能在 `Result` 上下文以外完成。

为什么日志不安全

在函数式编程中，你看不到太多的日志。这是因为函数式编程通常让日志记录没有用武之地。函数式程序是通过复合纯函数构建的，这意味着给定相同参数的函数始终返回相同的值，因此不会发生任何意外。另一方面，日志记录在命令式编程中普遍存在，因为在命令式程序中，你无法预测给定输入的输出。日志记录就像在说“我不知道这段程序在这个时候可能生成什么，所以我会把它写到一个日志文件中。如果一切顺利，我不需要这个日志文件，但是如果出现问题，我可以查看日志，看看程序在此时的状态。”真是无稽之谈。

在函数式编程中，不需要这样的日志。如果所有函数都正确（一般这一点都可以得到证明），你不需要知道中间状态。此外，在命令式程序中记录日志通常是有条件的，这意味着一些记录日志的代码只会在非常罕见和未知的状态下执行。这样的代码通常未经测试。如果你曾经看到，在 `INFO` 模式下运行良好的 Java 命令式程序在 `TRACE` 模式下运行时突然挂掉，那么你一定明白我的意思。

练习 13.2

在第 7 章中，你在 `Result` 类型中编写了一个 `forEachOrException` 方

法，它在 `Empty` 和 `Success` 中的工作方式类似于 `forEach`，额外还会返回一个 `Result.Empty`，而在 `Failure` 类中返回一个 `Result.Success<Exception>`。

编写一个 `forEachOrFail` 方法，它将返回带有异常消息的 `Result<String>`，而非异常本身。

请注意，这两个方法不是函数式的。虽然返回了一个值，但它们可能会有副作用。

答案 13.2

`Empty` 的实现除了返回 `Empty` 以外什么也没有：

```
public Result<String> forEachOrFail(Effect<T> c) {
    return empty();
}
```

`Success` 的实现应用作用并返回 `Empty`：

```
public Result<String> forEachOrFail(Effect<T> e) {
    e.apply(this.value);
    return empty();
}
```

`Failure` 的实现只返回一个包含异常或其消息的 `Success`：

```
public Result<String> forEachOrFail(Effect<T> c) {
    return success(exception.getMessage());
}

public Result<RuntimeException> forEachOrException(Effect<T> c) {
    return success(exception);
}
```

这些方法虽然不是函数式的，但它们大大简化了 `Result` 值的使用：

```
public class ResultTest {

    public static void main(String... args) {

        Result<Integer> ra = Result.success(4);
        Result<Integer> rb = Result.success(0);

        Function<Integer, Result<Double>> inverse = x -> x != 0
            ? Result.success((double) 1 / x)
            : Result.failure("Division by 0");

        Result<Double> rt1 = ra.flatMap(inverse);
        Result<Double> rt2 = rb.flatMap(inverse);
    }
}
```



```

System.out.print("Inverse of 4: ");
rt1.forEachOrFail(System.out::println).forEach(ResultTest::log);

System.out.print("Inverse of 0: ");
rt2.forEachOrFail(System.out::println).forEach(ResultTest::log);
}

private static void log(String s) {
    System.out.println(s);
}
}

```

该程序将打印如下内容：

```

Inverse of 4: 0.25
Inverse of 0: Division by 0

```

13.2 读取数据

到目前为止，你只处理了输出。如你所见，一旦计算完结果，当程序结束时便会输出数据。这允许函数式地编写大部分程序，从而得到该范式带来的所有效益。只有输出部分不是函数式的。我也说过，可以通过向其他程序发送数据来完成输出，但是你还没有看到如何将数据输入到你的程序中。现在我们来做一些。

稍后我们会看一下函数式地输入数据的方式。但是首先，就像输出那样，我们将讨论如何以一个整洁（虽然不是函数式而是命令式）的方式来输入数据，并相得益彰地匹配函数式的部分。

13.2.1 从控制台读取

例如，你将以命令式的方式从控制台读取数据，允许通过保证程序的确定性来进行测试。你将用到的方法类似于在第 12 章中对随机数发生器所做的操作。

你将首先开发一个读取整数和字符串的示例。清单 13.2 展示了你需要实现的接口。

清单 13.2 输入数据的接口

```

public interface Input {

    Result<Tuple<String, Input>> readString();

    Result<Tuple<Integer, Input>> readInt();
}

```

readInt 和 readString 方法将分别输入一个整数和一个字符串

```

default Result<Tuple<String, Input>> readString(String message) {
    return readString();
}

default Result<Tuple<Integer, Input>> readInt(String message) {
    return readInt();
}

```

这些方法允许你以参数的形式传递消息，这可能有助于提示用户，但是提供的默认实现会忽略该消息

你可以为这个接口编写一个具体的实现，但是首先你要编写一个抽象的（因为你也许会希望从其他来源中读取数据，如文件）。你要把公共代码放在抽象类中，并让每种类型的输入继承它。清单 13.3 展示了这个实现。

清单 13.3 AbstractReader 的实现

```

import com.fpinjava.common.Result;
import com.fpinjava.common.Tuple;
import java.io.BufferedReader;

public class AbstractReader implements Input {

    protected final BufferedReader reader;

    protected AbstractReader(BufferedReader reader) {
        this.reader = reader;
    }

    @Override
    public Result<Tuple<String, Input>> readString() {
        try {
            String s = reader.readLine();
            return s.length() == 0
                ? Result.empty()
                : Result.success(new Tuple<>(s, this));
        } catch (Exception e) {
            return Result.failure(e);
        }
    }

    @Override
    public Result<Tuple<Integer, Input>> readInt() {
        try {
            String s = reader.readLine();
            return s.length() == 0
                ? Result.empty()
                : Result.success(new Tuple<>(Integer.parseInt(s), this));
        } catch (Exception e) {

```

该类将通过一个读取器构建，允许不同的输入来源

readString 方法会从读取器中读取一行，如果该行为空，则返回 Result.Empty，如果取到了一些数据，则返回 Result.Success，如果出现问题，则返回 Result.Failure

```

        return Result.failure(e);
    }
}

```

为了从控制台中读取，你现在只需实现具体的类。这个类将负责提供读取器。此外，你还要重新实现接口的这两个默认方法为用户显示提示，如清单 13.4 所示。

清单 13.4 ConsoleReader 的实现

```

import com.fpinjava.common.Result;
import com.fpinjava.common.Tuple;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class ConsoleReader extends AbstractReader {

    protected ConsoleReader(BufferedReader reader) {
        super(reader);
    }

    @Override
    public Result<Tuple<String, Input>> readString(String message) {
        System.out.print(message + " ");
        return readString();
    }

    @Override
    public Result<Tuple<Integer, Input>> readInt(String message) {
        System.out.print(message + " ");
        return readInt();
    }

    public static ConsoleReader consoleReader() {
        return new ConsoleReader(new BufferedReader(
            new InputStreamReader(System.in)));
    }
}

```

重新实现两个默认方法来显示用户提示

静态工厂方法为底层抽象类提供了一个读取器

现在可以用你所学的和 ConsoleReader 类来编写一个从输入到输出的完整程序，参见清单 13.5。

清单 13.5 从输入到输出的完整程序

```

public class TestReader {

    public static void main(String... args) {

```

创建
了读
取器

```
Input input = ConsoleReader.consoleReader();
```

```
Result<String> rString =
    input.readString("Enter your name: ").map(t -> t._1);
```

```
Result<String> result =
    rString.map(s -> String.format("Hello, %s!", s));
```

```
> result.forEachOrFail(System.out::println)
    .forEach(System.out::println);
}
}
```

该行代表了该程序的业务部分。它可能是纯函数式的

上一节中的模式应用于输出结果或错误消息

调用了 `readString` 方法（带有用户提示），并返回一个 `Result<Tuple<String, Input>>`，它被映射以生成一个 `Result<String>`

并没有什么令人耳目一新。它相当于无处不在的“hello”程序，通常是大多数编程课中的第二个例子（就在“hello world”之后）！当然这只是一个例子。有意思的是，使它演变成为更有用的东西有多么容易。

练习 13.3

编写一个程序，反复要求用户输入一个整数 ID、一个名字和一个姓氏，之后将人员列表显示在控制台上。一旦用户输入空 ID，数据输入就会停止，然后显示输入数据的列表。

提示

你需要一个类来保存每一行数据，可使用清单 13.6 所示的 `Person` 类。

清单 13.6 `Person` 类

```
public class Person {

    private static final String FORMAT =
        "ID: %s, First name: %s, Last name: %s";

    public final int id;
    public final String firstName;
    public final String lastName;

    private Person(int id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

```

public static Person apply(int id, String firstName, String lastName) {
    return new Person(id, firstName, lastName);
}

@Override
public String toString() {
    return String.format(FORMAT, id, firstName, lastName);
}
}

```

可在 `ReadConsole` 类的 `main` 方法中实现这个答案。使用 `Stream.unfold` 方法生成一个 `person` 流。你可能会发现为输入数据对应到一个 `person` 创建一个单独的方法更容易，并为 `unfold` 的参数使用一个方法引用。这个方法可以有以下签名：

```
public static Result<Tuple<Person, Input>> person(Input input)
```

答案 13.3

答案很简单。考虑到有一个输入单人数据的方法，你可以创建一个 `person` 流并打印如下结果（在这种情况下忽略任何错误）：

```

Input input = ConsoleReader.consoleReader();
Stream<Person> stream = Stream.unfold(input, ReadConsole::person);
stream.toList().forEach(System.out::println);

```

`person` 方法就是你现在所需的全部。该方法只会询问 ID、名字和姓氏，生成三个 `Result` 实例，可以用先前章节中学到的解析式模式来复合：

```

public static Result<Tuple<Person, Input>> person(Input input) {
    return input.readInt("Enter ID:")
        .flatMap(id -> id._2.readString("Enter first name:")
            .flatMap(firstName -> firstName._2.readString("Enter last name:")
                .map(lastName -> new Tuple<>(Person.apply(id._1, firstName._1,
                    lastName._1), lastName._2)))));
}

```

请注意，解析式模式可能是函数式编程中最重要的模式之一，所以你确实要掌握它。其他语言如 `Scala` 或 `Haskell` 都为此提供了语法糖，但是 `Java` 却没有。它在伪代码中对应于类似下面这样的内容：

```

for {
    id in input.readInt("Enter ID:")
    firstName in id._2.readString("Enter first name:")
    lastName in firstName._2.readString("Enter last name:")
}

```

```
} return new Tuple<>(Person.apply(id._1, firstName._1,
                                lastName._1), lastName._2))
```

但你并不真的需要语法糖。首先，flatMap 语法可能一开始更难掌握，但它确实能显示发生了什么。

顺便提一句，许多程序员都知道如下的模式：

```
a.flatMap(b -> flatMap(c -> map(d -> getSomething(a, b, c, d))))
```

他们经常认为这是一系列以 map 结尾的 flatMaps。但绝非如此。以 map 还是 flatMap 结尾只取决于返回类型。最后一个方法（这里是 getSomething）经常会返回一个未包装的值（bare value），这就是为什么该模式以 map 结尾。但是如果 getSomething 返回一个上下文（如 Result），则该模式将变为如下所示：

```
a.flatMap(b -> flatMap(c -> flatMap(d -> getSomething(a, b, c, d))))
```

13.2.2 从文件中读取

这段程序的设计方式使它很容易能够改写为读取文件。FileReader 类与 ConsoleReader 非常相似。唯一的区别是静态工厂方法需要处理 IOException，所以它返回一个 Result<Input> 而不是一个未包装值。

清单 13.7 FileReader 的实现

```
import com.fpinjava.common.Result;
import java.io.*;

public class FileReader extends AbstractReader {

    private FileReader(BufferedReader reader) {
        super(reader);
    }

    public static Result<Input> fileReader(String path) {
        try {
            return Result.success(new FileReader(new BufferedReader(
                new InputStreamReader(new FileInputStream(new File(path))))));
        } catch (Exception e) {
            return Result.failure(e);
        }
    }
}
```

练习 13.4

编写一段类似于 ReadConsole 的 ReadFile 程序，只不过它是从文件中读取，其中的每一行都是一个条目。本书附带的代码 (<http://github.com/fpinjava/fpinjava>) 提供了示例文件。

提示

虽然它类似于 ReadConsole 程序，但你必须处理工厂方法返回 Result 的情况。尝试重用同一个 person 方法。

答案 13.4

答案在清单 13.8 中给出。请注意，在调用 person 方法前如何处理由工厂方法返回的 Result，允许你使用与 ConsoleReader 相同的方法。（你也可以使用不接收任何参数的 read 方法。）

清单 13.8 ReadFile 的实现

```
public class ReadFile {
    private static String path = "path to data file";

    public static void main(String... args) {
        Result<Input> rInput = FileReader.fileReader(path);
        Result<Stream<Person>> rStream =
            rInput.map(input -> Stream.unfold(input, ReadFile::person));
        rStream.forEachOrFail(stream -> stream.toList()
            .forEach(System.out::println)).forEach(System.out::println);
    }

    public static Result<Tuple<Person, Input>> person(Input input) {
        return input.readInt("Enter ID:")
            .flatMap(id -> id._2.readString("Enter first name:")
                .flatMap(firstName -> firstName._2.readString("Enter last name:")
                    .map(lastName -> new Tuple<>(Person.apply(id._1,
                        firstName._1, lastName._1), lastName._2))));
    }
}
```

将路径更改为你系统中的文件位置

在此处理 Result<Input>

13.2.3 检查输入

在以上答案中，你所采用的方法的好处之一是程序很容易被测试。当然，可以通过提供文件而不是让用户在控制台中输入来测试程序，不过很容易就能将

程序与另一个生成输入命令脚本的程序相连接。清单 13.9 展示了可用于测试的 `ScriptReader` 示例。

清单 13.9 允许使用输入命令列表的 `ScriptReader`

```
public class ScriptReader implements Input {

    private final List<String> commands;

    public ScriptReader(List<String> commands) {
        super();
        this.commands = commands;
    }

    public ScriptReader(String... commands) {
        super();
        this.commands = List.list(commands);
    }

    public Result<Tuple<String, Input>> readString() {
        return commands.isEmpty()
            ? Result.failure("Not enough entries in script")
            : Result.success(new Tuple<>(commands.headOption().getOrElse(""),
                new ScriptReader(commands.drop(1))));
    }

    @Override
    public Result<Tuple<Integer, Input>> readInt() {
        try {
            return commands.isEmpty()
                ? Result.failure("Not enough entries in script")
                : Integer.parseInt(commands.headOption().getOrElse("")) >= 0
                    ? Result.success(new Tuple<>(Integer.parseInt(
                        commands.headOption().getOrElse(""),
                        new ScriptReader(commands.drop(1))))
                    : Result.empty();
        } catch (Exception e) {
            return Result.failure(e);
        }
    }
}
```

清单 13.10 展示了一个使用 `ScriptReader` 类的示例。你将在本书附带的代码中找到单元测试的示例。

清单 13.10 使用 `ScriptReader` 输入数据

```
public class ReadScriptReader {
```



```
public static void main(String... args) {
    Input input = new ScriptReader(
        "0", "Mickey", "Mouse",
        "1", "Minnie", "Mouse",
        "2", "Donald", "Duck",
        "3", "Homer", "Simpson"
    );

    Stream<Person> stream =
        Stream.unfold(input, ReadScriptReader::person);
    stream.toList().forEach(System.out::println);
}

public static Result<Tuple<Person, Input>> person(Input input) {
    return input.readInt("Enter ID:")
        .flatMap(id -> id._2.readString("Enter first name:")
            .flatMap(firstName -> firstName._2.readString("Enter last name:")
                .map(lastName -> new Tuple<>(Person.apply(id._1, firstName._1,
                    lastName._1), lastName._2))));
}
}
```

13.3 真正的函数式输入/输出

迄今为止你已经学到的，对大多数 Java 程序员来说已经足够。分开程序的函数式部分与非函数式部分是必不可少的，也已经够用了。不过有趣的是，Java 程序可以变得更加函数式。

是否在生产代码中使用以下 Java 程序中的技术由你决定。也许不值得付出额外的复杂性。然而，学习这些技术既有用又有趣，所以你可以做出学以致用选择。

13.3.1 怎样才能让输入 / 输出是完全函数式的

这个问题有几个答案。最短的答案就是：不能。根据我们对函数式程序的定义，它是“一个除了返回值以外没有其他可观测到的作用的程序”，所以无法完成任何输入或输出。

但是许多程序并不需要完成任何输入或输出。例如，许多类库都属于这一类。类库是为了让其他程序使用而设计的程序。它们接收参数，并且返回基于参数计算出来的值。在本章的前两节中，你把程序分成三个部分，一个是输入，一个是输出，还有一个是完全函数式并充当一个类库的第三部分。

这个问题的另一种处理方式是编写这个类库的部分，并生成一个处理所有输入

和输出的另一个（非函数式）程序作为最终的返回值。这在概念上与惰性非常相似。你可以把输入和输出当作稍后发生的事情来处理，它将是一个单独的程序，由纯函数式程序返回。

13.3.2 实现纯函数式的输入 / 输出

在本节中，你将看到如何实现纯函数式的输入 / 输出。让我们先从输出开始。想象一下，你只是想要让控制台显示欢迎信息。现在，假设你已经知道要用于消息的名称。不要这么写：

```
static void sayHello(String name) {  
    System.out.println("Hello, " + name + "!");  
}
```

我们可以使 `sayHello` 方法返回一个程序，一旦运行，作用将会相同。你可以为此使用一个 `lambda` 和 `Runnable` 接口，如下所示：

```
static Runnable sayHello(String name) {  
    return () -> System.out.println("Hello, " + name + "!");  
}
```

你可以使用以下方法：

```
public static void main(String... args) {  
    Runnable program = sayHello("Georges");  
}
```

这段代码是纯函数式的。你可以认为它没有任何可见的东西，并且确实如此。它生成了一段程序，运行它可以生成想要的作用。该程序可以通过在其生成的 `Runnable` 上调用 `run` 方法来运行。返回的程序不是函数式的，但是无所谓。你的程序是函数式的。

这是作弊吗？不。思考一个用任何“函数式”语言编写的程序。到最后，它被编译为一个可执行程序，它完全不是函数式的，并且可以在你的计算机上运行。你所做的事情完全一样，只不过生成的程序看上去是用 `Java` 编写的。但其实不然。它是用你的程序正在构造的某种 `DSL`（领域特定语言）编写的。

要执行这个程序，你可以简单地编写

```
program.run();
```

请注意，大多数代码检查程序不会喜欢在一个 `Runnable` 上调用 `run` 的情况。这就是为什么在先前的章节中，你创建了 `Executable` 接口来做同样的事情。

在这里，你需要更强大的东西，因此你将创建一个名为 `IO` 的新接口。你将从一个 `run` 方法开始。在这个阶段，它与 `Runnable` 并没有什么不同：

```
public interface IO {
    void run();
}
```

假设你有以下三个方法：

```
static IO println(String message) {
    return () -> System.out.print(message);
}

static <A> String toString(Result<A> rd) {
    return rd.map(Object::toString).getOrElse(rd::toString);
}

static Result<Double> inverse(int i) {
    return i == 0
        ? Result.failure("Div by 0")
        : Result.success(1.0 / i);
}
```

你可以编写以下纯函数式程序：

```
IO computation = println(toString(inverse(3)));
```

该程序生成了可以稍后执行的另一段程序：

```
computation.run();
```

13.3.3 合并 IO

有了 `IO` 接口，你几乎可以构建任何程序，不过还仅仅是作为一个单元。能够合并这样的程序将会很有意思。你能使用的最简单的合并就是将两个程序合二为一。你将在以下练习中做到这点。

练习 13.5

在 `IO` 接口中创建一个方法，用于将两个 `IO` 实例合并为一个。该方法称为 `add`，并且会有一个默认实现。签名如下：

```
default IO add(IO io)
```

答案 13.5

答案只不过是返回一个实现了 `run` 的新 `IO`。它会首先执行当前 `IO`，然后才是参数 `IO`：

```
default IO add(IO io) {
  return () -> {
    IO.this.run();
    io.run();
  };
}
```

稍后你将会需要一个“什么也不干”的 `IO` 作为一些 `IO` 组合的中性元素。在 `IO` 接口中可以很容易地进行如下创建：

```
IO<Nothing> empty = () -> Nothing.instance;
```

借助这些新方法，你可以通过合并 `IO` 实例来创建更复杂的程序：

```
String name = getName();
```

```
IO instruction1 = println("Hello, ");
IO instruction2 = println(name);
IO instruction3 = println("!\n");
```

这三行什么也不打印。它们就像是 DSL 中的指令

```
IO script = instruction1.add(instruction2).add(instruction3);
script.run();
```

执行它

合并三个指令来
创建一段程序

当然，你可以简化这个过程：

```
println("Hello, ").add(println(name)).add(println("!\n")).run();
```

还可以从指令列表中创建一个程序：

```
List<IO> instructions = List.list(
  println("Hello, "),
  println(name),
  println("!\n")
);
```

这看起来像一个命令式的程序吗？事实上正是如此。可以使用右折叠来“编译它”：

```
IO program = instructions.foldRight(IO.empty(), io -> io::add);
```

或左折叠：

```
IO program = instructions.foldLeft(IO.empty(), acc -> acc::add);
```

你可以看到为什么需要一个“什么也不干”的实现。最后，程序可以照常运行：

```
program.run();
```

13.3.4 用 IO 处理输入

目前，你的 IO 类型只能处理输出。为了使其能够处理输入，一个必要的变更是用输入值的类型将其泛型化，以便它可以用于处理这个值。新的泛型 IO 类型如下所示：

```
public interface IO<A> {  <—— IO 接口是类型注解的

    A run();

    IO<Nothing> empty = () -> Nothing.instance;  <—— empty 实例没有类型参数，所以让它返回 Nothing 单例

    static <A> IO<A> unit(A a) {  <—— unit 方法接收一个未包装值，并在 IO 上下文中将其返回
        return () -> a;
    }
}
```

如你所见，IO 接口创建计算上下文的方式与 Option、Result、List、Stream、State 等类型相同。它同样具有返回 empty 实例的方法，以及在上下文中置入未包装值的方法。

为了处理 IO 值的计算，你现在需要像 map 和 flatMap 这样的方法来将函数绑定到 IO 上下文中。

练习 13.6

在 IO<A> 中定义一个 map 方法，以从 A 到 B 的函数为参数，并返回一个 IO，使其成为 IO 接口中的 default 实现。

答案 13.6

实现如下，它将该函数应用于 this 的值，并将结果返回到新的 IO 上下文中：

```
default <B> IO<B> map(Function<A, B> f) {
```

```
return () -> f.apply(this.run());
}
```

练习 13.7

编写一个 flatMap 方法，以一个从 A 到 IO 的函数为参数，并返回一个 IO。

提示

不用担心可能的栈溢出问题，稍后会再处理它。

答案 13.7

将函数应用于运行 this 这个 IO 所得到的值，将得到 IO<IO>。你需要展平这个结果（只要运行它即可轻易完成），如下所示：

```
default <B> IO<B> flatMap(Function<A, IO<B>>> f) {
    return () -> f.apply(this.run()).run();
}
```

如你所见，这是递归。刚开始它不是一个问题，因为只有一个递归步骤，但是如果你要链式调用大量的 flatMap，它可能会成为一个问题。

要查看你的新方法，请使用如下 Console 类，参见清单 13.11。

清单 13.11 Console 类

```
import com.fpinjava.common.Nothing;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Console {

    private static BufferedReader br =
        new BufferedReader(new InputStreamReader(System.in));

    public static IO<String> readLine(Nothing nothing) {
        return () -> {
            try {
                return br.readLine();
            } catch (IOException e) {
                throw new IllegalStateException((e));
            }
        };
    }

    public static IO<Nothing> printLine(Object o) {
```

readLine 方法以
Nothing 为参数并
返回 IO<String>

你重新抛出一个包装在运行时异常中的任何异常。记着这可不是 readLine 方法抛出的

println 方法以 Object 为
参数并返回 Nothing

```

return () -> {
    System.out.println(o.toString());
    return Nothing.instance;
};
}

```

表示为字符串的对象参数被打印到屏幕上。请记住，println 方法不进行打印。它返回一个 lambda，当它被执行时才会真正打印

值得注意的是，这两个方法都是纯函数式的。它们不会抛出任何异常，也不会从控制台读取或打印。它们只返回做这些事情的程序。

要在实际中看到这一点，可以运行以下示例程序，如清单 13.12 所示。

清单 13.12 以纯函数式的方式读取并打印到控制台

```

public class Main {
    public static void main(String... args) {
        IO<Nothing> script = sayHello();
        script.run();
    }

    private static IO<Nothing> sayHello() {
        return Console.println("Enter your name: ")
            .flatMap(Console::readLine)
            .map(Main::buildMessage)
            .flatMap(Console::println);
    }

    private static String buildMessage(String name) {
        return String.format("Hello, %s!", name);
    }
}

```

sayHello 方法返回一段程序

这段程序可以通过调用 run 来执行

这些行是指令，你从中构建程序

13.3.5 扩展 IO 类型

使用 IO 类型，你可以通过纯函数式的方式创建不纯的程序（含作用的程序）。但在这个阶段，这些程序只允许我们读取和打印一个元素，正如你的 Console 类。你可以通过添加指令来扩展你的 DSL 并创建控制结构，如循环和条件。

首先，你将实现一个循环，与 for 索引循环类似。这种形式将采用以迭代次数和待重复的 IO 为参数的 repeat 方法。

练习 13.8

在具有以下签名的 IO 接口中实现 repeat 为一个静态方法：

```
static <A> IO<List<A>> repeat(int n, IO<A> io)
```

提示

你应该创建一个 IO 实例的集合以表示每次迭代，然后通过合并这些 IO 实例来折叠该集合。你为此需要比 add 方法更为强大的东西。开始实现具有以下签名的 map2 方法：

```
static <A, B, C> IO<C> map2(IO<A> ioa, IO<B> iob,
                             Function<A, Function<B, C>> f)
```

答案 13.8

map2 方法可以如下实现：

```
static <A, B, C> IO<C> map2(IO<A> ioa, IO<B> iob,
                             Function<A, Function<B, C>> f) {
    return ioa.flatMap(a -> iob.map(b -> f.apply(a).apply(b)));
}
```

这是无处不在的解析式模式的一个简单应用。使用这种方法，你可以如下轻松实现 repeat：

```
static <A> IO<List<A>> repeat(int n, IO<A> io) {
    return Stream.fill(n, () -> io)
        .foldRight(() -> unit(List.list()), ioa -> sioLa -> map2(ioa,
                                                                    sioLa.get(), a -> la -> List.cons(a, la)));
}
```

请注意，你用 Stream.fill() 方法创建了一个流，它具有以下签名：

```
public static <T> Stream<T> fill(int n, Supplier<T> elem)
```

它返回一个 Stream，包含 n （惰性求值）个 T 实例。

这看起来也许有点复杂，但一部分原因是由于印刷而折行，另一部分是由于为了优化而将它编写为一行。它等价于：

```
static <A> IO<List<A>> repeat(int n, IO<A> io) {
    Stream<IO<A>> stream = Stream.fill(n, () -> io);
    Function<A, Function<List<A>, List<A>>> f = a -> la -> List.cons(a, la);
    Function<IO<A>, Function<Supplier<IO<List<A>>>, IO<List<A>>>> g =
        ioa -> sioLa -> map2(ioa, sioLa.get(), f);
    Supplier<IO<List<A>>> z = () -> unit(List.list());
    return stream.foldRight(z, g);
}
```

如果你用一个 IDE，那么找到类型相对比较容易。例如在 IntelliJ 中，你只需将

鼠标指针置于一个引用上，同时按住 Ctrl 键即可显示该类型。

通过这些方法，你现在可以如下编写：

```
IO program = IO.repeat(3, sayHello());
```

这样你将会得到一段程序，对应于像 sayHello(3) 那样调用以下方法：

```
private static void sayHello(int n) throws IOException {

    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

    for (int i = 0; i < n; i++) {
        System.out.println("Enter your name: ");
        String name = br.readLine();
        System.out.println(buildMessage(name));
    }
}
```

然而，非常重要的区别是，调用 sayHello(3) 会及早执行作用三次，而 IO.repeat(3, sayHello()) 只会简单地返回一个（未计算的）程序，只有当其 run 方法被调用时才会同样处理。

可以定义许多其他控制结构。你可以在从 <http://github.com/fpinjava/fpinjava> 下载的代码中找到示例。清单 13.13 展示了一个示例，使用 when 和 doWhile 方法的行为与命令式 Java 中的 if 和 while 完全相同。

清单 13.13 使用 IO 来包装命令式编程

```
public class Main {

    public static void main(String... args) throws IOException {

        IO program = program(buildMessage,
                               "Enter the names of the persons to welcome:");
        program.run();
    }

    public static IO<Nothing> program(Function<String, IO<Boolean>> f,
                                       String title) {

        return IO.sequence(
            Console.println(title),
            IO.doWhile(Console.readLine(), f),
            Console.println("bye!")
        );
    }

    private static Function<String, IO<Boolean>> buildMessage =
```

```

    name -> IO.when(name.length() != 0,
        () -> IO.unit(String.format("Hello, %s!", name))
    ).flatMap(Console::println);
}

```

这个例子并不是说建议你像这样编程。仅为输入和输出使用 IO 类型，而在函数式编程中进行所有的计算当然更好。毕竟，如果你选择学习函数式编程，那么不应该在函数式编码中实现一个命令式语言。但将它作为一个练习来了解其工作原理还是挺有意思的。

13.3.6 使 IO 类型栈安全

你可能没有注意到在先前的练习中，有些 IO 方法使用栈的方式与递归方法相同。例如，如果重复次数过高，则 repeat 方法将溢出栈。多少算是“太高”取决于栈的大小以及运行方法返回程序时的栈空间占用程度。（目前，我希望你明白，调用 repeat 方法不会将栈击穿，只有运行它返回的程序时才有可能）

练习 13.9

为了试验击穿栈，创建一个 forever 方法，以一个 IO 为参数并返回一个在无限循环中执行该参数的新 IO。相应的签名如下：

```
static <A, B> IO<B> forever(IO<A> ioa)
```

答案 13.9

它的难度和它的用处一样小！你要做的全部是使构造的程序无限递归。请注意，forever 方法本身不应该是递归的，只有返回的程序才应该是。答案是使用一个 Supplier，并通过一个在该 Supplier 上执行 get 的 IO 来 flatMap 参数 IO：

```

static <A, B> IO<B> forever(IO<A> ioa) {
    Supplier<IO<B>> t = () -> forever(ioa);
    return ioa.flatMap(x -> t.get());
}

```

该方法可以如下使用：

```

public static void main(String... args) {
    IO program = IO.forever(IO.unit("Hi again!")
        .flatMap(Console::println));
    program.run();
}

```

它将在数千次迭代之后把栈击穿。注意它相当于以下内容：

```
IO.forever(Console.println("Hi again!")).run();
```

如果你不明白为什么它会把栈击穿，请思考以下伪代码（不能编译！），其中 `t` 变量被替换成了相应的表达式：

```
static <A, B> IO<B> forever(IO<A> ioa) {
    return ioa.flatMap(x -> (() -> forever(ioa)).get());
}
```

现在让我们把递归调用替换为相应的代码：

```
static <A, B> IO<B> forever(IO<A> ioa) {
    return ioa.flatMap(x -> (() -> ioa.flatMap(x -> (() -
        > forever(ioa)).get()).get()));
}
```

你可以继续无限递归。（记住，你不应该尝试编译这段代码！）你可能会注意到，对 `flatMap` 的调用会被嵌套，导致每次调用都会把当前的状态推送到栈上，于是会在几千次之后把栈击穿。与命令式代码依次地执行指令不同，`flatMap` 方法被递归调用了。

为了使 IO 栈安全，你可以使用与第 4 章中使用的相同技术来创建栈安全的递归方法和函数。首先，你需要表示程序的三个状态：

- `Return` 表示完成的计算，意味着你只需返回结果即可。
- `Suspend` 表示一个暂停的计算，在恢复当前的计算之前需要应用一些作用。
- `Continue` 表示一个状态，程序必须首先应用一个子计算，然后才继续下一个。

这些状态将表示为清单 13.14 所示的三个类。

注意 清单 13.14 至清单 13.16 组成了一个整体。它们不应该分开使用，而应该一起使用。

清单 13.14 这三个类需要使 IO 栈安全

```
final static class Return<T> implements IO<T> {
    public final T value;

    protected Return(T value) {
```

这个 value 将由计算返回

```

    this.value = value;
}

@Override
public boolean isReturn() {
    return true;
}

@Override
public boolean isSuspend() {
    return false;
}
}

final static class Suspend<T> implements IO<T> {

    public final Supplier<T> resume;

    protected Suspend(Supplier<T> resume) {
        this.resume = resume;
    }

    @Override
    public boolean isReturn() {
        return false;
    }

    @Override
    public boolean isSuspend() {
        return true;
    }
}

final static class Continue<T, U> implements IO<T> {

    public final IO<T> sub;
    public final Function<T, IO<U>>> f;

    protected Continue(IO<T> sub, Function<T, IO<U>>> f) {
        this.sub = sub;
        this.f = f;
    }

    @Override
    public boolean isReturn() {
        return false;
    }

    @Override
    public boolean isSuspend() {
        return true;
    }
}

```

用于决定 IO 性质的辅助方法。相应的抽象方法在 IO 接口中声明

该 Supplier 充当了一个无参函数，应用（副）作用并返回值得

用于决定 IO 性质的辅助方法。相应的抽象方法在 IO 接口中声明

该 IO 首先执行，生成一个值得

通过将此函数应用于返回值来继续计算

用于决定 IO 性质的辅助方法。相应的抽象方法在 IO 接口中声明

```

    return false;
}
}

```

需要对外围的 IO 接口进行一些修改，如清单 13.15 和清单 13.16 所示。

清单 13.15 IO 栈安全版本的变化

```

import com.fpinjava.common.*;
import static com.fpinjava.common.TailCall.ret;
import static com.fpinjava.common.TailCall.sus;

public abstract class IO<A> {

    protected abstract boolean isReturn();
    protected abstract boolean isSuspend();

    private static IO<Nothing> EMPTY =
        new IO.Suspend<>(() -> Nothing.instance);

    public static IO<Nothing> empty() {
        return EMPTY;
    }

    public A run() {
        return run(this);
    }

    public A run(IO<A> io) {
        return run_(io).eval();
    }

    private TailCall<A> run_(IO<A> io) {
        ... // see listing 13.16
    }

    public <B> IO<B> map(Function<A, B> f) {
        return flatMap(f.andThen(Return::new));
    }

    @SuppressWarnings("unchecked")
    public <B> IO<B> flatMap(Function<A, IO<B>> f) {
        return (IO<B>) new Continue<>(this, f);
    }

    static <A> IO<A> unit(A a) {
        return new IO.Suspend<>(() -> a);
    }
}

```

IO 类型现在是一个抽象类

empty IO 现在是一个 Suspend。它是 private 的，增加了相应的 public 访问器

run 方法现在简单地调用辅助方法 run(this)

run(this) 方法依次调用将返回一个 TailCall 的 run_ 辅助方法

run_ 辅助方法如清单 13.16 所示

现在 map 方法被定义为将 flatMap 应用于 f 和 Return 构造函数的复合函数

flatMap 方法返回一个转换为 IO<A> 的 Continue

unit 方法返回一个 Suspend

清单 13.16 栈安全的 run 方法

该方法返回一个 TailCall，它将由发起调用的方法求值

```
private TailCall<A> run_(IO<A> io) {
```

```
    if (io.isReturn()) {
```

```
        return ret(((Return<A>) io).value);
```

```
    } else if (io.isSuspend()) {
```

```
        return ret(((Suspend<A>) io).resume.get());
```

```
    } else {
```

```
        Continue<A, A> ct = (Continue<A, A>) io;
```

```
        IO<A> sub = ct.sub;
```

```
        Function<A, IO<A>> f = ct.f;
```

```
    } if (sub.isReturn()) {
```

```
        return sus(() -> run_(f.apply(((Return<A>) sub).value)));
```

```
    } else if (sub.isSuspend()) {
```

```
        return sus(() -> run_(f.apply(((Suspend<A>) sub).resume.get())));
```

```
    } else {
```

```
        Continue<A, A> ct2 = (Continue<A, A>) sub;
```

```
        IO<A> sub2 = ct2.sub;
```

```
        Function<A, IO<A>> f2 = ct2.f;
```

```
        return sus(() -> run_(sub2.flatMap(x ->
```

```
            f2.apply(x).flatMap(f)));
```

```
    }
```

```
}
```

```
}
```

如果 sub 是一个 Return，则递归调用该方法，并传入应用其包含的函数的结果

如果接收到的 IO 是一个 Return，则计算结束

如果接收到的 IO 是一个 Suspend，则在返回恢复值之前执行其包含的作用

如果接收到的 IO 是一个 Continue，则读取包含的子 IO

如果 sub 是一个 Continue，则提取它所包含的 IO (sub2)，并且 flatMap 其与 sub，从而创建链式调用

如果 sub 是一个 Suspend，则应用其包含的函数，如果有的话还可能生成函数的作用

栈安全的新版本可以如清单 13.17 所示的那样使用。

清单 13.17 新的 Console 类使用了栈安全的版本

```
public class Console {
```

```
    private static BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

```
    public static IO<String> readLine(Nothing nothing) {
```

```
        return new IO.Suspend<>(() -> {
```

```
            try {
```

```
                return br.readLine();
```

```
            } catch (IOException e) {
```

```
                throw new IllegalStateException((e));
```

```
            }
```

```
        });
```

```
    }
```

```

/**
 * readLine 函数的一种可能实现
 */
public static Function<Nothing, IO<String>> readLine_ = x -> new
    IO.Suspend<>(() -> {
        try {
            return br.readLine();
        } catch (IOException e) {
            throw new IllegalStateException((e));
        }
    });

/**
 * readLine 函数使用方法引用的一种简单实现
 */
public static Function<Nothing, IO<String>> readLine = Console::readLine;

/**
 * 无须提供 Nothing 即可调用 readLine 方法的一个简便的辅助手法
 */
public static IO<String> readLine() {
    return readLine(Nothing.instance);
}

public static IO<Nothing> printLine(Object s) {
    return new IO.Suspend<>(() -> println(s));
}

private static Nothing println(Object s) {
    System.out.println(s);
    return Nothing.instance;
}

public static IO<Nothing> printLine_(Object s) {
    return new IO.Suspend<>(() -> {
        System.out.println(s);
        return Nothing.instance;
    });
}

public static Function<String, IO<Nothing>> printLine_ =
    s -> new IO.Suspend<>(() -> {
        System.out.println(s);
        return Nothing.instance;
    });

public static Function<String, IO<Nothing>> printLine = Console::printLine;
}

```

现在，你可以使用 `forever` 或 `doWhile` 了，毫无栈溢出的风险。你也可以重

写 `repeat` 使其栈安全。我不会在这里展示新的实现，但你可以在随书附带的代码 (<http://github.com/fpinjava/fpinjava>) 中找到它。

请记住，这不是编写函数式程序的推荐方法。把它当作最终可以完成的一个例子，而别当作一个好的实践。另外需要注意的是，这里的“最终”适用于 Java 编程。通过对函数式更加友好的语言，你可以精心制作更为强大的程序。

13.4 总结

- 作用可以传递到 `List`、`Result` 和其他上下文中，以安全地应用于其值，而不是从这些上下文中提取值并在外部应用作用，如果没有值，可能会发生错误。
- 可以在 `Result` 类型内部处理 `success` 和 `failure` 这两种不同的作用。
- 可以用与第 12 章中生成随机数相同的方式来完成读取数据的操作。
- 可以用与从控制台或内存相同的方式通过 `Reader` 抽象来完成从文件读取。
- 可以通过 `IO` 类型获得更多函数式的输入 / 输出。
- `IO` 类型可以扩展为更通用的类型，这样便能够以函数式的方式通过构建延迟执行的程序来执行命令式任务。
- 通过使用让递归方法栈安全的相同技术可以使 `IO` 类型栈安全。

通过actor共享可变状态

本章要点

- 了解 actor 模型
- 使用异步消息
- 构建一个 actor 框架
- 开始使用 actor
- 优化 actor 的性能

在阅读本书时，你首先学到了函数式编程处理的一般是不可变数据，从而导致程序更安全、更可靠，并且易于设计和扩展。然后你学习了如何通过以状态为参数传递给函数来以函数式的方式处理可变状态。你还看到了这种技术的几个例子：

- 在生成随机数时传递发生器，允许增加可测试性。
- 以控制台为参数可以将函数式输出发送到屏幕，并从键盘接收输入。

这种技术可以广泛应用于许多领域。在命令式编程中，解析文件一般通过连续地改变表示解析结果的组件状态来处理。为了使这个过程与函数式编程兼容，你只需将状态作为附加参数传递给所有解析函数即可。还可以用相同的方式记录日志和

监控性能：你可以使该函数以日志文件为参数，并将变大的文件作为结果的一部分返回，而不是在每个函数中写入日志文件。

这种方法的好处是，它可以使你在访问资源时无须太关心同步和锁。但是这种安全性是通过防止数据共享获得的。这样很好是因为它迫使你找到其他更安全的方法。使用不可变列表并不会为共享这些列表的操作自动增加安全性。它只是阻止你共享可变状态而已。它允许你以与类似生成防御性副本相对应的方式伪装一个列表的变化，但是没有性能损失。这么做确实有用，但有时它不是你需要的。

假设你打算统计一个函数被调用的次数。在单线程应用程序中，你可以将计数器添加到函数参数中并将递增后的计数器作为结果的一部分返回。但是大部分命令式程序员宁愿以副作用的方式递增计数器。在只有一个线程时，这样做天衣无缝，不必用锁来防止潜在的并发访问。这和生活在荒岛上一样，如果你是唯一的居民，你的门确实不需要锁。

但是在多线程程序中，如何以安全的方式增加计数器，避免并发访问？答案通常是使用锁或是使操作成为原子操作，或两者兼有。

在函数式编程中，必须把共享资源当作一种作用，也就是说，每当访问共享资源时，你需要告别函数式的安全性，并把这个操作当作在第13章对输入/输出那样处理。这是否意味着你之后需要管理锁和同步？并非如此。正如你在前几章中学到的，函数式编程也与抽象到极致有关。共享可变状态可以这样来抽象，以便你可以使用它而不必考虑细节。使用 actor 框架是实现这个目标的一个办法。

与先前的章节不同，在这里你不会开发一个真正完整的 actor 框架。创建一个完整的 actor 框架是一项庞大的工作，你应该使用一个现成的框架。你将在此开发一个迷你的 actor 框架，让你感受到 actor 框架为函数式编程带来的影响。

14.1 actor模型

在 actor 模型中，多线程应用程序被分为基本上是单线程的组件，称为 actor。如果每个 actor 都是单线程的，则不需要使用锁或同步来共享数据。actor 通过作用与其他 actor 通信，就像这样的通信是输入/输出一样。这意味着 actor 依靠一种机制来序列化它们收到的消息。（这里的序列化意味着一个接一个地依次处理消息，可别与 Java 的序列化混淆。）由于这种机制，它们可以一次处理一个消息，而不必再

操心并发访问它们的资源。因此，actor 系统可以被视为通过作用相互通信的一系列函数式程序。每个 actor 都可以是单线程的，所以内部没有并发访问资源。在框架内部抽象了并发。

14.1.1 异步消息

作为消息处理的一部分，actor 可以向其他 actor 发送消息。消息异步发送，即不需要等待答案。一旦消息发送完成，发送方可以继续工作，主要是依次处理它接收到的消息队列。当然，处理消息队列意味着需要管理对队列的一些并发访问。但这样的管理已经抽象到 actor 框架中了，所以作为程序员的你，不必担心这一点。

当然可能需要回复消息。假设一个 actor 负责长时间的计算。客户端可以通过在处理计算时继续自己的工作来从异步中获益。但是一旦完成了计算，客户端就需要有一种接收结果的办法。这可以简单地通过让负责计算的 actor 回调其客户端，并以异步方式再次发送结果。请注意，客户端可能是原来的发送方，虽然并不总是需要这样。

14.1.2 处理并行

actor 模型允许任务通过使用 manager actor 并行，manager actor 负责将任务分解为子任务并将它们分发给多个 worker actor。当一个 worker actor 将结果返回给 manager 时，它将被赋予一个新的子任务。该模型提供了其他并行化模型无法做到的一个优点：只要子任务列表不为空，任何 worker actor 都不会闲置。缺点就是 manager actor 不会参与计算。但是在实际应用中通常不会有什么明显的区别。

对于某些任务而言，在接收到子任务时，它们的结果可能需要重新排序。在这种情况下，manager actor 可能会将结果发送给负责这个工作的特定 actor。你将在 14.2.3 节中看到这样一个例子。在小规模的程序中，manager 自己就可以处理这个任务。在图 14.1 中，这个 actor 被称为 Receiver（接收者）。

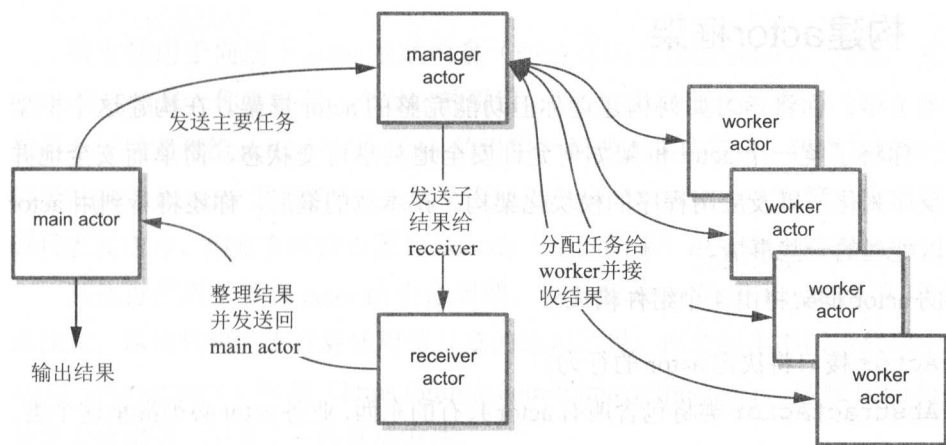


图 14.1 main actor 生成主要任务并将其发送给 manager actor，任务随即被拆分为几个子任务，由几个 worker actor 并行处理。子结果将发送回 manager，再被传递给 receiver。子结果整理完成后，receiver 将最终结果发送给 main actor。

14.1.3 处理 actor 状态变化

actor 可以无状态（不可变）也可以有状态，这意味着它们应该根据收到的消息来改变自身状态。例如，一个 synchronizer actor 接收到的计算结果可能必须在使用前重新排序。

假设你有一个必须经过大量计算的数据列表以提供结果列表。简而言之，这就是一个映射。它可以通过将列表拆分为几个子列表并将它们分配给 worker actor 来并行处理。但是，不能保证 worker actor 按照把这些工作交给它们的顺序来完成。对任务进行编号是重新同步结果的一种办法。当 worker 返回结果时，它会添加相应的任务号码，以便 receiver 将结果存放在优先队列中。这不仅允许自动排序，还提供了用异步流处理结果的可能。每当 receiver 接收到结果时，它会比较任务编号与预期编号。如果匹配，则将结果传递给客户端，然后查找优先队列，看看第一个可用结果是否对应于新的预期任务编号。如果再次匹配，则继续出队处理直到不再匹配为止。如果接收到的结果与预期结果编号不匹配，则将其添加到优先队列中。

在这样的设计中，receiver actor 必须处理两份可变的数据：优先队列和预期结果编号。这是否意味着 actor 必须使用可变属性？这不是什么大不了的事，但由于 actor 是单线程的，所以其实没必要。如你所见，通用的状态变化过程可以包含并抽象属性变化的处理，允许程序员只用不可变数据。

14.2 构建 actor 框架

在本节中，你将学习如何构建迷你但功能完整的 actor 框架。在构建这个框架的同时，你将了解一个 actor 框架如何允许安全地共享可变状态，简单而安全地并行化和反序列化，以及应用程序的模块化架构。在本章的最后，你还将看到用 actor 框架可以处理的一些事情。

你的 actor 框架将由 4 个组件构成：

- Actor 接口将决定 actor 的行为。
- AbstractActor 类将包含所有 actor 共有的东西。业务 actor 必须继承这个类。
- ActorContext 将作为访问 actor 的一种方式。在你的实现中，这个组件将非常精简，主要用于访问 actor 的行为。在这么小的实现中，这个组件并不是必需的，但是大多数严谨的实现都会使用这样的组件。例如，它允许搜索可用的 actor。
- MessageProcessor 接口将是你为需要处理接收到的消息的任意组件实现的接口。

14.2.1 actor 框架的限制

如我所言，你在这里创建的实现是极简的；把它当作理解和练习使用 actor 模型的方式。你会缺少一个真正的 actor 系统的许多（大部分？）功能，尤其是那些与 actor 上下文相关的功能。另一个简化是每个 actor 将被映射到单个线程。在一个真正的 actor 系统中，actor 被映射到线程池，允许在几十个线程上运行数千乃至数百万个 actor。

你的实现的另一个限制是大多数 actor 框架允许以透明的方式处理分布式 actor，这意味着你可以使用在不同机器上运行的 actor，而无须担心通信问题。这当然使得 actor 框架成为构建可扩展应用程序的理想方式。我们不会涉及这个方面。

14.2.2 设计 actor 框架接口

首先，你需要定义构成 actor 框架的接口。最重要的当然是定义拥有几种方法的 Actor 接口。这个接口的主要方法是

```
void tell(T message, Result<Actor<T>> sender)
```

该方法用于向这个 actor 发送消息（即持有该方法的 actor）。当然，要向一个 actor 发送消息，你需要有一个对它的引用。（这与真正的 actor 框架不同，真正的框架不会发送消息给 actor，而是 actor 的引用、代理或其他替身。没有这种增强功能，将无法向远程 actor 发送消息。）这个方法以 `Result<Actor>` 为第二个参数。它应该代表发送方，但它有时被设置为 `nobody`（空的结果）或一个不同的 actor。

其他方法用于管理 actor 的生命周期，以方便 actor 的使用，如清单 14.1 所示。请注意，这段代码并不打算使用前几章的练习结果，而会用本书附带代码中可用的 `fpinjava-common` 模块 (<https://github.com/fpinjava/fpinjava>)。这些代码与练习的答案大致相同，但多了一些其他方法。

清单 14.1 Actor 接口

```
public interface Actor<T> {
```

```
    static <T> Result<Actor<T>> noSender() {
        return Result.empty();
    }
```

noSender 方法是一个辅助方法，提供了一个 `Result<Actor>` 类型的 `Result.Empty`

```
    Result<Actor<T>> self();
```

self 方法返回对该 actor 的引用

```
    ActorContext<T> getContext();
```

getContext 方法允许你访问 actor 上下文

```
    default void tell(T message) {
        tell(message, self());
    }
```

```
    void tell(T message, Result<Actor<T>> sender);
```

这是一个通过不必指定发送方来简化发送消息的快捷方法

```
    void shutdown();
```

```
    default void tell(T message, Actor<T> sender) {
        tell(message, Result.of(sender));
    }
```

这是另一个快捷方法，允许你通过 actor 的引用而非 `Result<Actor>` 来发送消息

```
    enum Type {SERIAL, PARALLEL}
}
```

shutdown 方法允许你告诉 actor 它应该自行终止。在你的迷你框架中，它会允许你终止 actor 线程

在某些特定情况下，Actor 可以配置为多线程

清单 14.2 展示了另外两个必要的接口：`ActorContext` 和 `MessageProcessor`。

清单 14.2 ActorContext 和 MessageProcessor 接口

```

public interface ActorContext<T> {
    void become(MessageProcessor<T> behavior);
    MessageProcessor<T> getBehavior();
}

public interface MessageProcessor<T> {
    void process(T t, Result<Actor<T>> sender);
}

```

become 方法允许 actor 通过注册一个新的 MessageProcessor 来改变其行为

这个方法允许访问 actor 的行为

MessageProcessor 接口只有一个方法，表示处理一条消息

这里最重要的就是 ActorContext 接口。become 方法允许 actor 改变其行为，即它处理消息的方式。如你所见，一个 actor 的行为看起来像一个作用，以待处理的消息和发送方为参数。

在应用程序的生命周期中，每个 actor 的行为都允许改变。一般来说，这种行为的变化由修改 actor 的状态所引起，用新的行为替代原来的行为。一旦你看到实现，便会更清楚。

14.2.3 AbstractActor 的实现

AbstractActor 的实现代表了所有 actor 共用实现的一部分。所有的消息管理操作是共用的，由 actor 框架提供，因此你只需实现业务部分即可。AbstractActor 的实现如清单 14.3 所示。

清单 14.3 AbstractActor 的实现

```

import com.fpinjava.common.Result;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.RejectedExecutionException;

public abstract class AbstractActor<T> implements Actor<T> {

    private final ActorContext<T> context;
    protected final String id;
    private final ExecutorService executor;

    public AbstractActor(String id, Actor.Type type) {
        super();
        this.id = id;
        this.executor = type == Type.SERIAL
            ? Executors.newSingleThreadExecutor(new DaemonThreadFactory())

```

初始化底层的
ExecutorService

context 属性 初始化为一个新的
ActorContext

默认行为
委托给
onReceive
方法

为了改变其行为, ActorContext
只需注册新的行为即可。这是
变化发生的地方, 但它被框架
隐藏了

onReceive 方法将持有业务
处理部分, 并将由 API 的
用户实现

tell 方法即 actor 如何接收消
息。它是同步的, 以确保一
次只处理一个消息

当接收到一条消息时, 它由
actor 上下文返回的当前行为
处理

```

        : Executors.newCachedThreadPool(new DaemonThreadFactory());

this.context = new ActorContext<T>() {
    private MessageProcessor<T> behavior =
        AbstractActor.this::onReceive;

    @Override
    public synchronized void become(MessageProcessor<T> behavior) {
        this.behavior = behavior;
    }

    @Override
    public MessageProcessor<T> getBehavior() {
        return behavior;
    }
};

public abstract void onReceive(T message, Result<Actor<T>> sender);

public Result<Actor<T>> self() {
    return Result.success(this);
}

public ActorContext<T> getContext() {
    return this.context;
}

@Override
public void shutdown() {
    this.executor.shutdown();
}

public synchronized void tell(final T message, Result<Actor<T>> sender) {
    executor.execute(() -> {
        try {
            context.getBehavior().process(message, sender);
        } catch (RejectedExecutionException e) {
            /*
             * 这可能是正常情况, 表示由于 actor 被终止,
             * 所有等待中的任务都被取消了
             */
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    });
}
}

```


请注意，如果 actor 将用于单线程，Executor 由单线程 executor 初始化，这也是绝大多数的情况；如果用于多线程，则由缓存线程池初始化。线程池通过守护线程工厂创建，以允许在主线程终止时自动关闭。

你的 actor 框架现在已经完成，但是我先前提过，这并不是生产代码。它只是一个用于向你展示 actor 框架如何工作的最小型例子。

14.3 开始使用actor

现在你有了一个可用的 actor 框架，已经到了将它应用到一些具体问题上的时候了。当多个线程应该共享一些可变状态时，actor 是有用的，就像线程生成计算结果并将其传递给另一个线程以进行下一步处理。通常，这种可变状态共享是通过将值存储在共享的可变属性中来实现的，这就意味着锁和同步。我们先来看一个最小的 actor 示例，它可以被视为 actor 的“Hello, World!”。然后我们会学习一个更完整的应用程序，用一个 actor 将任务分配给其他并行工作的 actor。

第一个例子是用于测试 actor 的最小的经典示例。它由两名乒乓球选手和一名裁判员组成。当一个以整数表示的球给予一个玩家时，游戏开始。然后每个玩家将球发送给另一个玩家，直到发生了 10 次为止，这时球被送回给裁判员。

14.3.1 实现乒乓示例

首先你要实现裁判员。你需要做的全部就是创建一个 actor，实现其 onReceive 方法。在这个方法中显示一条消息：

```
Actor<Integer> referee =
    new AbstractActor<Integer>("Referee", Actor.Type.SERIAL) {
        @Override
        public void onReceive(Integer message, Result<Actor<Integer>> sender) {
            System.out.println("Game ended after " + message + " shots");
        }
    };
```

接下来，你必须创建两名选手。因为有两个实例，所以你不会将它们创建为匿名类。创建一个 Player 类，参见清单 14.4。

清单 14.4 Player actor

sound 字符串是选手收到球时要显示的信息 (“Ping” 或者 “Pong”)

每名选手都用裁判员的引用创建，以便选手在游戏结束时可以将球送回给裁判

```
static class Player extends AbstractActor<Integer> {
    private final String sound;
    private final Actor<Integer> referee;

    public Player(String id, String sound, Actor<Integer> referee) {
        super(id, Actor.Type.SERIAL);
        this.referee = referee;
        this.sound = sound;
    }

    @Override
    public void onReceive(Integer message, Result<Actor<Integer>> sender) {
        System.out.println(sound+"-"+ message);
        if (message >= 10) {
            referee.tell(message, sender);
        } else {
            sender.forEachOrFail(actor -> actor.tell(message + 1, self()))
                .forEach(ignore -> referee.tell(message, sender));
        }
    }
}
```

这是 actor 的
“业务”部分

如果游戏结束，将球送回给裁判

否则，如果另一名选手存在的话，将球送回给对方。如果没有别的选手，向裁判员注册一个问题

随着 Player 类的创建，你的程序可以定稿了。但是你需要一种方法来保持应用程序运行直到游戏结束。不然的话，应用程序的主线程将在游戏开始时终止，选手们就没有机会玩游戏了。可以通过使用计数信号量(semaphore)来实现，如清单 14.5 所示。

清单 14.5 乒乓示例

```
private static final Semaphore semaphore = new Semaphore(1);
public static void main(String... args) throws InterruptedException {
    Actor<Integer> referee =
        new AbstractActor<Integer>("Referee", Actor.Type.SERIAL) {
        @Override
        public void onReceive(Integer message, Result<Actor<Integer>> sender) {
            System.out.println("Game ended after " + message + " shots");
            semaphore.release();
        }
    };
    Actor<Integer> player1 = new Player("Player1", "Ping", referee);
```

用 1 个许可证创建计数信号量

当游戏结束时，计数信号量被释放，使一个新的许可证变得可用，从而允许主线程恢复

```
Actor<Integer> player2 = new Player("Player2", "Pong", referee);

semaphore.acquire();
player1.tell(1, Result.success(player2));
semaphore.acquire();
}
```

当前线程获得一个可用的许可证，游戏开始

主线程尝试获得新的许可证。因为还没有可用的许可证，它将阻塞直到计数信号量被释放为止

恢复时，主线程终止。所有 actor 线程都是守护线程，所以它们也会自动停止

该程序显示以下输出：

```
Ping - 1
Pong - 2
Ping - 3
Pong - 4
Ping - 5
Pong - 6
Ping - 7
Pong - 8
Ping - 9
Pong - 10
Game ended after 10 shots
```

14.3.2 一个更严谨的例子：并行运行一个计算

现在是时候来看一个比较严谨的 actor 框架示例了：并行运行一个计算。为了模拟一个长时间运行的计算，你将从 0 到 30 之间选择一个随机数列表，并使用一个很慢的算法来计算相应的斐波那契值。应用程序将由三种 actor 组成：一个 Manager，负责创建一定数量的 worker actor 并对它们分配任务；几个 worker 的实例；还有一个客户端，它将在主程序类中实现为一个匿名的 actor。清单 14.6 展示了这些类中最简单的 Worker actor。

清单 14.6 负责运行部分计算的 Worker actor

```
import com.fpinjava.actors.AbstractActor;
import com.fpinjava.actors.Actor;
import com.fpinjava.common.Result;
import com.fpinjava.common.TailCall;

public class Worker extends AbstractActor<Integer> {

    public Worker(String id, Type type) {
```

```

    super(id, type);
}

@Override
public void onReceive(Integer message, Result<Actor<Integer>> sender) {
    sender.forEach(a -> a.tell(fibo(message), self()));
}

private static int fibo(int number) {
    return fibo_(0, 1, number).eval();
}

private static TailCall<Integer> fibo_(int acc1, int acc2, int x) {
    if (x == 0) {
        return TailCall.ret(1);
    } else if (x == 1) {
        return TailCall.ret(acc1 + acc2);
    } else {
        return TailCall.sus(() -> fibo_(acc2, acc1 + acc2, x - 1));
    }
}

```

当 Worker 收到一个数字时，它做出的反应是计算相应的斐波那契值并将其发送回调用方

fibo 方法使用一个尾递归辅助方法

你用非常低效的算法来创建持续时间较长的任务

如你所见，这个 actor 没有状态。它计算结果并将其发送回已经接收过引用的发送方。请注意，这个 actor 可能与调用方不同。因为选择的数字是 0 到 30 之间的随机数，所以计算结果所需的时间变化非常大。这样便模拟了执行可变时长的任务。与第 8 章中的自动并行化示例不同，除非没有更多任务要启动，否则所有的线程 / actor 将持续忙碌，直到整个计算完成。

Manager 类有点复杂，清单 14.7 展示了类的构造函数和初始化过的属性。

清单 14.7 Manager 类的构造函数和属性

Manager 存储了它的客户端的引用，将向其发送计算结果

```

import com.fpinjava.actors.AbstractActor;
import com.fpinjava.actors.Actor;
import com.fpinjava.actors.MessageProcessor;
import com.fpinjava.common.*;

public class Manager extends AbstractActor<Integer> {
    private final Actor<Result<List<Integer>>> client;
    private final int workers;
    private final List<Tuple<Integer, Integer>> initial;

```

存储要使用的 worker 的数量

初始列表会是整型元组的列表，持有要处理的数字 (1) 和列表 (2) 中的位置

managerFunction 是 Manager 的核心，决定了它能做什么。每当 manager 从 worker 接收到结果时，将应用这个函数

workList 是当所有的 worker actor 都被分配了第一个任务以后，剩下的待执行任务列表

```
private final List<Integer> workList;
private final List<Integer> resultList;
private final Function<Manager, Function<Behavior,
    Effect<Integer>>> managerFunction;
```

resultList 将保存计算结果

```
public Manager(String id, List<Integer> list,
    Actor<Result<List<Integer>>> client, int workers) {
    super(id, Type.SERIAL);
    this.client = client;
    this.workers = workers;
    Tuple<List<Integer>, List<Integer>> splitLists =
        #list.splitAt(this.workers);
```

用 worker 的数量拆分待处理的值列表，以获取初始任务列表和剩余任务列表

```
    this.initial = splitLists._1.zipWithPosition();
    this.workList = splitLists._2;
    this.resultList = List.list();
```

将 workList 设置为剩余的任务

```
    managerFunction = manager -> behavior -> i -> {
        List<Integer> result = behavior.resultList.cons(i);
        if (result.length() == list.length()) {
            this.client.tell(Result.success(result.reverse()));
        } else {
            manager.getContext()
                .become(new Behavior(behavior.workList
                    .tailOption()
                    .getOrElse(List.list()), result));
        }
    };
```

否则，调用上下文的 become 方法来更改 Manager 的行为。这种行为的改变实际上在这里是状态的改变。新的行为通过 workList 的 tail 和当前结果列表（接收到的值已经被添加到了这里）来创建

如果 resultList 的长度等于输入列表的长度，则计算结束，反转结果并将其发送给客户端

在收到结果后，它会被添加到从 manager 行为中取得的结果列表中

resultList 被初始化为一个空列表

代表 manager 工作的 manager 函数是 manager 自身、它的行为和收到的消息 (i) 的一个柯里化函数，收到的消息就是子任务的结果

初始任务列表（计算斐波那契值的数字）将按照其元素的位置被压缩。位置（从 0 到 n 的数字）将仅用于命名 worker actor 为 0 到 n

如你所见，如果计算结束，把结果添加到结果列表中并发送给客户端。否则，把结果添加到当前的结果列表中。在传统程序中，通过改变由 Manager 持有的结果列表来完成。这里也是一样，不过有两个区别：

- 结果列表被存储在行为中。
- 行为和列表都不会改变。相反，创建一个新行为，并修改上下文以将旧行为替换为新行为。但是，你不必处理这个改变。就你而言，一切都是不可变的，因为 actor 框架抽象了这个改变。

清单 14.8 展示了实现为内部类的 Behavior 类。

清单 14.8 Behavior 内部类允许你抽象 actor 的改变

```
class Behavior implements MessageProcessor<Integer> {

    private final List<Integer> workList;
    private final List<Integer> resultList;

    private Behavior(List<Integer> workList, List<Integer> resultList) {
        > this.workList = workList;
        this.resultList = resultList;
    }

    @Override
    public void process(Integer i, Result<Actor<Integer>> sender) {
        managerFunction.apply(Manager.this).apply(Behavior.this).apply(i);
        sender.forEach(a -> workList.headOption().forEachOrFail(x ->
            a.tell(x, self()).forEach(x -> a.shutdown())));
    }
}
```

Behavior 通过 workList（在调用构造函数之前已经删除了 head）和 resultList（结果已添加于此）构造

在接收到消息时要调用的 process 方法首先将 managerFunction 应用于接收到的消息。然后它将下一个任务（workList 的 head）发送给发送方（将处理它的 worker actor），或者如果 workList 为空，则它只需通知 worker actor 关闭即可

这涵盖了 Manager 的主要部分。其余部分由主要用于开始工作的工具方法所组成，参见清单 14.9。

清单 14.9 Manager 的工具方法，用于开始处理

```
public class Manager extends AbstractActor<Integer> {
```

```
...
```

```

public void start() {
    onReceive(0, self());
    initial.sequence(this::initWorker)
        .forEachOrFail(this::initWorkers)
        .forEach(this::tellClientEmptyResult);
}

private Result<Executable> initWorker(Tuple<Integer, Integer> t) {
    return Result.success(() ->
        new Worker("Worker " + t._2, Type.SERIAL).tell(t._1, self()));
}

private void initWorkers(List<Executable> lst) {
    lst.forEach(Executable::exec);
}

private void tellClientEmptyResult(String string) {
    client.tell(Result.failure(string + " caused by empty input list."));
}

@Override
public void onReceive(Integer message, Result<Actor<Integer>> sender) {
    getContext().become(new Behavior(workList, resultList));
}

```

为了开始, Manager 向自己发送消息。由于行为尚未被初始化, 所以什么消息都没关系

然后创建和初始化 worker

这个方法创建一个 Executable, 可以创建一个 worker actor

这个方法执行 actor 的创建

这是 Manager 的初始行为。作为它的初始化的一部分, 它切换行为, 从包含剩余任务的 workList 和空的 resultList 开始

如果有错误, 则通知客户端

理解这一点很重要: onReceive 方法表示了当 actor 收到第一个消息时会做什么。当 worker 将结果发送给 manager 时, 不会调用这个方法。

程序的最后一部分如清单 14.10 所示。WorkersExample 类表示应用程序的客户端代码。但与 Manager 和 Worker 不同, 它不是 actor。相反, 它有一个 actor。这是一个实现的选择。没有必须选择某个方案的特殊原因。但是为了要获得结果, 客户端 actor 是必要的。

清单 14.10 客户端应用程序

创建信号量以允许主线程等待 actor 完成它们的工作

```

public class WorkersExample {

    private static final Semaphore semaphore = new Semaphore(1);

```

初始化任务数量

通过随机生成 0 到 30 之间的数字来创建任务列表

在这里设置 worker actor 的数量

```
private static int listLength = 200_000;
private static int workers = 8;
private static final List<Integer> testList =
    SimpleRNG.doubles(listLength, new SimpleRNG.Simple(3))
        ._1.map(x -> (int) (x * 30)).reverse();

public static void main(String... args) throws InterruptedException {
    semaphore.acquire();
    final AbstractActor<Result<List<Integer>>> client =
        new AbstractActor<Result<List<Integer>>>("Client", Actor.Type.SERIAL) {
            @Override
            public void onReceive(Result<List<Integer>> message,
                                   Result<Actor<Result<List<Integer>>>> sender) {
                message.forEachOrFail(WorkersExample::processSuccess)
                    .forEach(WorkersExample::processFailure);
                semaphore.release();
            }
        };
    final Manager manager =
        new Manager("Manager", testList, client, workers);
    manager.start();
    semaphore.acquire();
    private static void processFailure(String s) {
        System.out.println(s);
    }

    public static void processSuccess(List<Integer> lst) {
        System.out.println("Result: " + lst.takeAtMost(40));
    }
}
```

客户端唯一的责任是处理结果或任何发生的错误

客户端在收到结果时释放信号量

再次获取信号量以等待工作完成

实例化 manager 并启动

创建客户端 actor 为匿名类

程序启动时获取信号量

你可以使用不同长度的任务列表和不同数量的 worker actor 来运行这段程序。在我的八核 Linux 主机上运行长度为 200 000 的任务，结果如下：

- 1 个 worker actor : 3.5 秒
- 2 个 worker actor : 1.5 秒
- 3 个 worker actor : 1.1 秒
- 4 个 worker actor : 0.8 秒
- 6 个 worker actor : 0.8 秒
- 8 个 worker actor : 0.8 秒
- 16 个 worker actor : 0.8 秒

这些数字当然不是非常精确，但是它们表明了使用与可用内核数相应数量的线程并没有用。程序显示的结果如下（仅显示前 40 个结果）：

```
Input: [0, 11, 28, 13, 20, 5, 15, 8, 24, 19, 12, 7, 11, 4, 18, 20, 26,
        21, 15, 21, 29, 16, 15, 8, 22, 11, 26, 1, 22, 13, 25, 3, 13, 24, 29,
        10, 7, 26, 24, 1, NIL]
Time: 797
Result: [1, 8, 28657, 34, 196418, 34, 987, 987, 1597, 832040, 28657,
        17711, 987, 377, 1, 17711, 196418, 377, 10946, 4181, 5, 6765, 144,
        21, 75025, 233, 832040, 89, 144, 75025, 514229, 21, 377, 1, 10946,
        3, 17711, 196418, 144, 1597, NIL]
```

如你所见，我们遇到问题了！

14.3.3 重新排序结果

你可能已经注意到了结果不正确。在看第 3 个和第 5 个随机值（28 和 20）及其相应的结果（28657 和 196418）时非常明显。你还可以比较 4 和 6 的值与结果。当参数值为 13 和 5 时，结果都是 34。请注意，如果你在自己的计算机上运行该程序，则会获得不同的结果。

在这里发生的是，并非所有的任务都需要相同的执行时间。我选择了以这种方式执行的计算，所以一些任务（计算小的值）迅速返回，而其他任务（计算更大的值）则需要更长的时间。因此，返回值的顺序不正确。

为了解决这个问题，你需要按照相应参数的顺序对结果进行排序。一种办法是使用你在第 11 章中开发的 `Heap` 数据类型。你可以对每个任务编号，并将这个编号用于优先队列中的优先级。

你需要做的第一件事是修改 `worker actor` 的类型。它们必须使用整型元组而不是整型：一个整型代表参数或计算，另一个代表任务编号。清单 14.11 展示了

Worker 类中的相应修改。

清单 14.11 Worker actor 追踪任务编号

```
public class Worker extends AbstractActor<Tuple<Integer, Integer>> {
    public Worker(String id, Type type) {
        super(id, type);
    }

    @Override
    public void onReceive(Tuple<Integer, Integer> message,
        Result<Actor<Tuple<Integer, Integer>>> sender) {
        sender.forEach(a -> a.tell(new Tuple<>(fibo(message._1),
            message._2), self()));
    }
    ...
}
```

类型参数从整型改成 Tuple<Integer, Integer>

修改 onReceive 方法的签名以反映新的 actor 类型

返回的消息改为
包含了任务编号

请注意，任务编号是元组的第二个元素。这既不易读又难以记忆，因为任务编号和计算参数的类型相同（Integer）。这不应该在现实生活中发生，因为你应该为任务使用一个特定的类型。但是如果你愿意，也可以用特定的类型而非 Tuple 来包装任务和任务编号，例如具有数字属性的 Task 类型。

Manager 类中的更改更多。首先，你需要修改该类的类型还有 workList 和 result 属性的类型：

```
public class Manager extends AbstractActor<Tuple<Integer, Integer>> {
    ...
    private final List<Tuple<Integer, Integer>> workList;
    private final Heap<Tuple<Integer, Integer>> resultHeap;
```

这些属性在构造函数中如下进行初始化：

```
Tuple<List<Tuple<Integer, Integer>>, List<Tuple<Integer, Integer>>>
    splitLists = list.zipWithPosition().splitAt(this.workers);
this.initial = splitLists._1;
this.workList = splitLists._2;
this.resultHeap = Heap.empty((t1, t2) -> t1._2.compareTo(t2._2));
```

workList 现在包含了元组（如上一个示例中的初始列表的情况），结果是元组的优先队列（Heap）。请注意，该 Heap 由基于元组的第二个元素做比较的

Comparator 初始化。使用包含任务和任务编号的 Task 类型将允许你将此类型设为 Comparable, 这样 Comparator 就没有用了。(我留下这个优化给你做练习。)

当然, managerFunction 也不一样:

```
private final Function<Manager, Function<Behavior, Effect<Tuple<Integer, Integer>>>> managerFunction;
```

它在构造函数中如下进行初始化:

```
managerFunction = manager -> behavior -> i -> {
    Heap<Tuple<Integer, Integer>> result = behavior.resultHeap.insert(i);
    if (result.length() == list.length()) {
        this.client.tell(Result.success(result.toList()
            .map(x -> x._1).reverse()));
    } else {
        ...
    }
};
```

接收的结果现在插入 Heap 中

计算完成后, 在返回给客户端之前把 Heap 转换为列表

需要修改 Behavior 内部类以反映 actor 类型的变化:

```
class Behavior implements MessageProcessor<Tuple<Integer, Integer>> {
    private final List<Tuple<Integer, Integer>> workList;
    private final Heap<Tuple<Integer, Integer>> resultHeap;

    private Behavior(List<Tuple<Integer, Integer>> workList,
        Heap<Tuple<Integer, Integer>> resultHeap) {
        this.workList = workList;
        this.resultHeap = resultHeap;
    }

    @Override
    public void process(Tuple<Integer, Integer> i,
        Result<Actor<Tuple<Integer, Integer>>> sender) {
        managerFunction.apply(Manager.this).apply(Behavior.this).apply(i);
        ...
    }
}
```

Behavior 类的类型参数现在是 Tuple<Integer, Integer>

result 的类型现在是 Heap<Tuple<Integer, Integer>>

workList 的类型现在是 List<Tuple<Integer, Integer>>

相应地修改构造函数的签名

修改 process 方法的签名以反映参数类型的变化

在 Manager 类的其余部分中仍然有一些小的修改。start 方法需要修改：

```
public void start() {
    onReceive(new Tuple<>(0, 0), self());
    initial.sequence(this::initWorker)
        .forEachOrFail(this::initWorkers)
        .forEach(this::tellClientEmptyResult);
}
```

start 消息的类型必须与 Manager actor 的类型参数相匹配

Worker 的初始化过程也略有不同：

```
private Result<Executable> initWorker(Tuple<Integer, Integer> t) {
    return Result.success(() -> new Worker("Worker " + t._2,
        Type.SERIAL).tell(new Tuple<>(t._1, t._2), self()));
}
```

最后，修改 onReceive 方法：

```
@Override
public void onReceive(Tuple<Integer, Integer> message,
    Result<Actor<Tuple<Integer, Integer>>> sender) {
    getContext().become(new Behavior(workList, resultHeap));
}
```

现在结果能以正确的顺序显示了。但是又多了一个新问题：现在所需的计算时间是 1 个 worker actor 15 秒，4 个 worker actor 13 秒。发生什么了？

答案很简单：瓶颈是 Heap。Heap 数据结构不适合排序。当保持较少的元素数量时，它有良好的性能，但是在这里你把全部 20 万个结果都插入堆中，每次插入都对整个数据集进行排序。这样效率当然不高。

14.3.4 解决性能问题

显然，这种低效不是一个实现问题，而是一个选用恰当工具的问题。你可以通过存储所有结果并在计算结束时进行一次排序来获得更好的性能，需要使用正确的工具进行排序。

另一种选择是修复你的实现。当前设计的一个问题是，不仅需要很长时间往 Heap 中插入，而且它是由 Manager 线程完成的，因此，不必在 worker actor 的计算完成时分配任务给它们，Manager 只需让它们等到插入堆操作完成即可。为插入 Heap 使用单独的 actor 是一种可行的方案。

但有时候更好的办法是选用恰当的工具。事实上，同步地消费结果可能不是一个需求。如果不是，你只是增加了一个使问题更难解决的隐式需求。一种可能是将

结果单独传递给客户端。这样的话，只有当结果乱序时才用 Heap，以防它变得太大。

事实上，这种用法正是为优先队列量身打造的。考虑到这一点，你可以将 Receiver actor 添加到你的程序中，参见清单 14.12。

清单 14.12 Receiver actor，负责异步接收结果

```

Receiver 类是一个由待接收的数据类型参
数化的 actor: Integer

Receiver 客户端是
由 List<Integer> 类
型参数化的 actor

public class Receiver extends AbstractActor<Integer> {

    private final Actor<List<Integer>> client;
    private final Function<Receiver, Function<Behavior,
        Effect<Integer>>>> receiverFunction;

    public Receiver(String id, Type type, Actor<List<Integer>> client) {
        super(id, type);
        this.client = client;
        receiverFunction = receiver -> behavior -> i -> {
            if (i == -1) {
                this.client.tell(behavior.resultList.reverse());
                shutdown();
            } else {
                receiver.getContext()
                    .become(new Behavior(behavior.resultList.cons(i)));
            }
        };

        @Override
        public void onReceive(Integer i, Result<Actor<Integer>> sender) {
            getContext().become(new Behavior(List.list(i)));
        }

        class Behavior implements MessageProcessor<Integer> {

            private final List<Integer> resultList;

            private Behavior(List<Integer> resultList) {
                this.resultList = resultList;
            }

            @Override
            public void process(Integer i, Result<Actor<Integer>> sender) {
                receiverFunction.apply(Receiver.this).apply(Behavior.this).apply(i);
            }
        }
    }
}

```

Receiver 函数接收一个 Integer。如果它是 -1，意味着计算完成，它会将结果发送给客户端并关闭自己

否则，它会通过将结果添加到结果列表来改变其行为

初始的 onReceive 实现包括将 actor 的行为替换为使用包含第一个结果的新列表

这个 Behavior 持有当前的结果列表

主类 (WorkersExample) 与上面的例子并没有太大的区别。唯一不同的是增加了 Receiver:

```
public static void main(String... args) throws InterruptedException {
    semaphore.acquire();
    final AbstractActor<List<Integer>> client =
        new AbstractActor<List<Integer>>("Client", Actor.Type.SERIAL)
    {
        @Override
        public void onReceive(List message, Result<Actor<List<Integer>>> sender) {
            System.out.println("Result: " + message.takeAtMost(40));
            semaphore.release();
        }
    };

    final Receiver receiver = new Receiver("Receiver", Actor.Type.SERIAL, client);
    final Manager manager = new Manager("Manager", testList, receiver, workers);
    manager.start();
    semaphore.acquire();
}
```

Worker actor 与上一个示例完全相同, 你只需处理持有最重要变更的 Manager 类。第一个变更是 Manager 将会有有一个类型为 Actor<Integer> 的客户端, 并将记录任务列表的长度:

```
private final Actor<Integer> client;
...
private final int limit;
...
public Manager(String id, List<Integer> list, Actor<Integer> client,
               int workers) {
    super(id, Type.SERIAL);
    this.client = client;
    this.workers = workers;
    this.limit = list.length() - 1;
}
```

另请注意, 现在客户端是 Receiver, 所以它的类型为 Actor<Integer>, 依次异步接收结果。

managerFunction 当然不一样了:

这个函数现在调用 streamResult 方法, 返回一个 Tuple3。第一个元素是结果 Heap, 结果就添加到这里。第二个元素是下一个预期的结果编号, 第三个元素是按预期顺序排列的结果 List

```
managerFunction = manager -> behavior -> t -> {
    Tuple3<Heap<Tuple<Integer, Integer>>, Integer, List<Integer>> result =
        streamResult(behavior.resultHeap.insert(t),
                     behavior.expected, List.list());
```

```

result._3.reverse().foreach(this.client::tell);
if (result._2 > limit) {
    this.client.tell(-1);
} else {
    manager.getContext()
        .become(new Behavior(bbehavior.workList.tailOption()
            .getOrElse(List.list()), result._1, result._2));
}
};

```

如果所有的任务都执行完毕，客户端就会发送一个特殊的终止代码

如你所见，大部分工作都是在 streamResult 方法中完成的：

```

private Tuple3<Heap<Tuple<Integer, Integer>>, Integer,
List<Integer>> streamResult(Heap<Tuple<Integer, Integer>> result,
    int expected, List<Integer> list) {
    Tuple3<Heap<Tuple<Integer, Integer>>, Integer, List<Integer>> tuple3 =
        new Tuple3<>(result, expected, list);
    Result<Tuple3<Heap<Tuple<Integer, Integer>>, Integer,
        List<Integer>>> temp = result.head().flatMap(head ->
        result.tail().map(tail -> head._2 == expected
            ? streamResult(tail, expected + 1, list.cons(head._1))
            : tuple3));
    return temp.getOrElse(tuple3);
}

```

也许这个方法令人费解，但这只是因为 Java 中的类型标记是如此冗长的缘故。streamResult 方法接收结果 Heap、下一个预期的任务编号和一开始为空的整型列表为参数：

- 如果结果堆的 head 与预期的任务结果编号不同，则什么也不用做，直接将三个参数作为 Tuple3 返回。
- 如果结果堆的 head 与预期的任务结果编号相符，则从堆中将其删除并添加到列表中。然后递归调用该方法，直到 head 不再匹配为止，于是结果列表按照预期的顺序构建，而其他结果将留在堆中。

通过这种处理方式，堆总是保持较小。例如，当计算 200 000 个任务时，堆的大小为 121 封顶。有 12 次超过 100，而 95% 以上的时间它都小于 2。

图 14.2 展示了在 Manager 的角度上接收结果的整个过程。

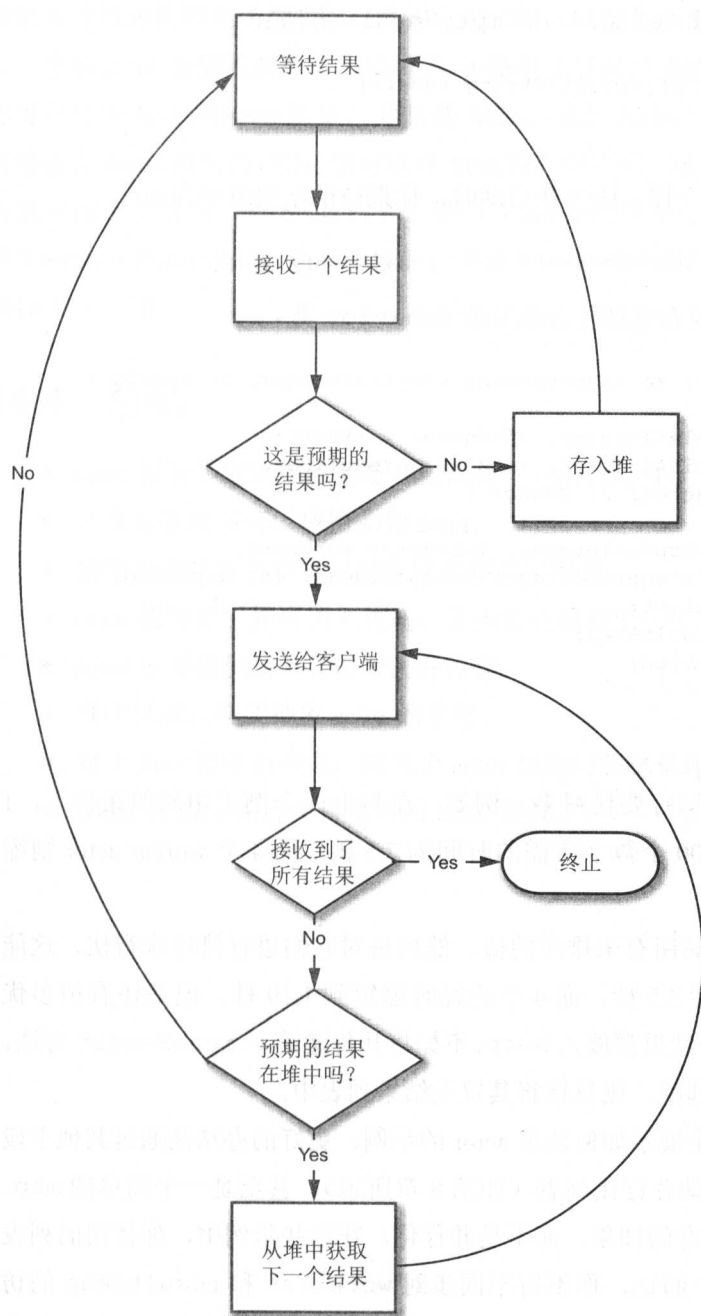


图 14.2 Manager 接收到一个结果，并将其存储在堆中（如果不是预期的编号）或将其发送给客户端。在后一种情况下，它会查看堆，看看是否已经接收过了下一个预期结果。

根据客户端的类型修改 `tellClientEmptyResult` 方法：

```
private void tellClientEmptyResult(String ignore) {
    client.tell(-1);
}
```

`onReceive` 方法不太一样，因为在启动时，你期待的结果编号为 0：

```
getContext().become(new Behavior(workList, resultHeap, 0));
```

最后一个改变是现在保存预期任务编号的 `Behavior` 类：

```
class Behavior implements MessageProcessor<Tuple<Integer, Integer>> {

    private final List<Tuple<Integer, Integer>> workList;
    private final Heap<Tuple<Integer, Integer>> resultHeap;
    private final int expected; // Change

    private Behavior(List<Tuple<Integer, Integer>> workList,
                     Heap<Tuple<Integer, Integer>> resultHeap, int expected) {
        this.workList = workList;
        this.resultHeap = resultHeap;
        this.expected = expected;
    }

    ...
}
```

通过这些修改，程序运行要快得多。例如，在与上一个例子相同的条件下，1 个 `worker actor` 处理 200 000 个数字所需的时间为 7.5 秒，而 4 个 `worker actor` 则缩短到 5.3 秒。

这个流程显然不如存储所有未排序的值，然后再对它们进行排序来得快，这能使一个 `actor` 的耗时缩短到 3.5 秒，而 4 个的耗时缩短到 1.19 秒。但是还有很多优化空间。例如，与其把每个结果都放入 `Heap`，不如将其传递给 `streamResult` 方法，如果它与预期的任务编号匹配，则直接将其置于结果列表中。

无论如何，这只是一个展示如何使用 `actor` 的示例。更好的办法是通过其他手段来解决这类问题，例如自动并行化列表（如第 8 章所示），甚至是一个简单的 `map`。`actor` 主要用于共享可变状态的抽象，而不是并行化。在这些示例中，你使用的列表在任务间共享。没有 `actor` 的话，你不得不同步对 `workList` 和 `resultHeap` 的访问来处理并发。`actor` 允许你在框架中抽象同步和变更。如果你看看自己写的业务代码（除了 `actor` 框架本身以外），你会发现没有可变的数据，因此无须在意同步，并且没有线程饥饿或死锁的风险。虽然它们不是函数式的，但是 `actor` 提供了一个很

棒的方式以使代码的函数式部分能够一起工作，以抽象的方式共享可变状态。

你的 actor 框架实在是过于精简了，不要用于任何严谨的代码。对于那样的需求，你可以使用 Java 的 actor 框架，尤其是 Akka。虽然 Akka 是用比 Java 更加函数式友好的语言 Scala 编写的，但它也可以在 Java 程序中使用。使用 Akka 时，除非你愿意，否则永远不会看到一行 Scala 代码。要了解 actor 的更多信息，尤其是 Akka，请参阅 Raymond Roestenburg、Rob Bakker 和 Rob Williams 编写的 *Akka in Action* (Manning, 2016 年) 一书。

14.4 总结

- actor 以异步方式接收消息并依次处理它们的组件。
- 共享可变状态可以被抽象为 actor。
- 抽象可变状态共享可以减少同步和并发问题。
- actor 模型基于异步消息传递，是函数式编程的一个绝佳补充。
- actor 模型提供简单又安全的并行化。
- 程序员通过框架抽象 actor 的变化。
- 对于 Java 程序员来说，有几个 actor 框架可尝试使用。
- Akka 是 Java 编程中最常用的 actor 框架之一。

15 以函数式的方式解决 常见问题

本章要点

- 使用断言
- 读取属性文件
- 适配命令式类库

现在你可以自由支配许多可以让你的程序员人生更加轻松的函数式工具了。但是只知道工具还不够。为了更加高效地进行函数式编程，你需要让它成为第二本能。你需要函数式的思维。最初你还会保留命令式的本能，并且很可能需要思考如何把命令式的方案转换为函数式的编码。当你在接触编程问题时，第一反应是在思考一个函数式的方案时（说不定把它转换为命令式还有些困难），你就成为一名精通函数式编程的程序员了。

为了达到这一阶段，除了练习以外别无他法。并且至少在 Java 世界里，由于大量常见问题的已知方案都是命令式的，了解一些常见问题并尝试用函数式的方式来解决，是一个不错的练习。

在网络上有大量以函数式的方式来解决数学问题的例子。这些例子非常有趣，但是它们有时适得其反地让程序员感觉函数式编程只适用于解决数学问题。更糟糕

的是，它导致一些人以为练习函数式编程必须要具有较高的数学能力。可并不是这样啊。对于解决数学问题而言，数学能力是必需的，但是你需要解决的大部分编程问题与数学无关，并且用函数式的方式解决经常会更简单。

在本章中，我们会看到在日常的职业生涯中程序员需要解决的一些常见问题，并且见识一下用函数式范式来处理它们会有什么不同。

15.1 使用断言来校验数据

Java 自 1.4 版起就有断言（assertion）了。断言用于检查不变量（invariant），如前置条件（precondition）、后置条件（post-condition）、流程控制条件（control-flow condition）和类条件（class condition）。函数式编程中一般是没有流程控制的，而类通常都是不可变的，所以要检查的只有前置条件和后置条件，由于相同的原因（不可变性和没有流程控制），包括检查方法和函数所接收的参数，以及在返回之前检查其结果。

有必要在以下偏函数中检查参数的值：

```
double inverse(int x) {  
    return 1.0 / x;  
}
```

该方法对任意输入都返回了一个恰当的值，除了 0 以外，它应该返回“无穷大”。由于对这个值很可能无从下手，所以你也许更倾向于以一种特定方式来处理它。在命令式编程里，你可以这样编写：

```
double inverse(int x) {  
    assert x != 0;  
    return 1.0 / x;  
}
```

但是在 Java 中你可以在运行时禁止断言，所以常用的技巧是通过 static 初始化来阻止程序启用断言：

```
static {  
    boolean assertsEnabled = false;  
    assert assertsEnabled = true;  
    if (!assertsEnabled) {  
        throw new RuntimeException("Asserts must be enabled!!!");  
    }  
}
```

这是 Oracle 给出的建议。当然，这样写更简单：

```
double inverse(int x) {  
    if (x == 0) throw new IllegalArgumentException("div. By 0");  
    return 1.0 / x;  
}
```

在函数式编程中，函数需要被转换为全函数，如下所示：

```
Result<Double> inverse(int x) {  
    return x == 0  
        ? Result.failure("div. By 0")  
        : Result.success(1.0 / x);  
}
```

这样就没有必要检查参数了，因为检查是函数实现的一部分。当然也没有必要检查返回值。

有一个经常需要检查的条件是参数不为 null。Java 为此提供了 `Objects.requireNonNull`。这个方法有许多重载，可以接收附加的错误消息，或是错误消息的 `Supplier`。这些方法有时很有用：

```
public static <T, U> Tuple<T, U> t(T t, U u) {  
    return new Tuple<>(Objects.requireNonNull(t), Objects.requireNonNull(u));  
}
```

但是在函数式编程中，更通用的断言形式是用一个特定条件来检验参数，并且在条件不匹配时返回一个 `Result.Failure`，否则返回一个 `Result.Success`。以 `Person` 类型的工厂方法为例：

```
public static Person apply(int id, String firstName, String lastName) {  
    return new Person(id, firstName, lastName);  
}
```

这个方法可以用于从数据库中获取的数据：

```
Person person = Person.apply(rs.getInt("personId"),  
                             rs.getString("firstName"), rs.getString("lastName"));
```

在这种情况下，你也许会想在调用 `apply` 方法之前校验数据。例如，你可能想要检查 ID 是正数，还有名和姓都不是 null 或空字符串，并以大写字母开头。在命令式 Java 里，可以通过使用断言方法来完成：

```

Person person = Person.apply(
    assertPositive(rs.getInt("personId"), "Negative id"),
    assertValidName(rs.getString("firstName"), "Invalid first name:"),
    assertValidName(rs.getString("lastName"), "Invalid last name:"));

private static int assertPositive(int i, String message) {
    if (i < 0) {
        throw new IllegalStateException(message);
    } else {
        return i;
    }
}

private static String assertValidName(String name, String message) {
    if (name == null || name.length() == 0 ||
        || name.charAt(0) < 65 || name.charAt(0) > 91) {
        throw new IllegalStateException(message);
    }
    return name;
}

```

在函数式编程中，你并不抛出异常，而是用像 `Result` 这样的特定上下文来处理错误。这种校验的类型被抽象到 `Result` 类型中。你需要做的全部就是编写校验函数，这也意味着只要编写方法并使用方法的引用即可。通用的校验函数可以被组织到一个特定的类中：

```

public class Assertion {
    public static boolean isPositive(int i) {
        return i >= 0;
    }

    public static boolean isValidName(String name) {
        return name != null && name.length() != 0
            && name.charAt(0) >= 65 && name.charAt(0) <= 91;
    }
}

```

接下来可以校验数据：

```

Result<Person> person =
    Result.of(Assertion::isPositive, getInt("personId"), "Negative id")
        .flatMap(id -> Result.of(Assertion::isValidName,
            getString("firstName"), "Invalid first name")
            .flatMap(firstName -> Result.of(Assertion::isValidName,
                getString("lastName"), "Invalid last name")
                .map(lastName -> Person.apply(id, firstName, lastName))));

```

但是你也可以通过在 `Assertion` 类中抽象更多流程来简化：

```

public static Result<Integer> assertPositive(int i, String message) {
    return Result.of(Assertion::isPositive, i, message);
}

public static Result<String> assertValidName(String name, String message) {
    return Result.of(Assertion::isValidName, name, message);
}

```

还可以如下创建一个 Person：

```

Result<Integer> rId = Assertion.assertPositive(getInt("personId"), "Negative id");
Result<String> rFirstName =
    Assertion.assertValidName(getString("firstName"), "Invalid first name");
Result<String> rLastName =
    Assertion.assertValidName(getString("lastName"), "Invalid first name");
Result<Person> person =
    rId.flatMap(id -> rFirstName
        .flatMap(firstName -> rLastName
            .map(lastName -> Person.apply(id, firstName, lastName))));

```

清单 15.1 展示了 Assertion 类和一些示例方法。

清单 15.1 函数式断言示例

```

public final class Assertion {
    private Assertion() {}

    public static <T> Result<T> assertCondition(T value,
                                                Function<T, Boolean> f) {
        return assertCondition(value, f,
            "Assertion error: condition should evaluate to true");
    }

    public static <T> Result<T> assertCondition(T value,
                                                Function<T, Boolean> f, String message) {
        return f.apply(value)
            ? Result.success(value)
            : Result.failure(message, new IllegalStateException(message));
    }

    public static Result<Boolean> assertTrue(boolean condition) {
        return assertTrue(condition,
            "Assertion error: condition should be true");
    }

    public static Result<Boolean> assertTrue(boolean condition,
                                                String message) {
        return assertCondition(condition, x -> x, message);
    }
}

```

```
public static Result<Boolean> assertFalse(boolean condition) {
    return assertFalse(condition,
        "Assertion error: condition should be false");
}

public static Result<Boolean> assertFalse(boolean condition,
    String message) {
    return assertCondition(condition, x -> !x, message);
}

public static <T> Result<T> assertNotNull(T t) {
    return assertNotNull(t, "Assertion error: object should not be null");
}

public static <T> Result<T> assertNotNull(T t, String message) {
    return assertCondition(t, x -> x != null, message);
}

public static Result<Integer> assertPositive(int value) {
    return assertPositive(value,
        String.format("Assertion error: value %s must be positive", value));
}

public static Result<Integer> assertPositive(int value, String message) {
    return assertCondition(value, x -> x > 0, message);
}

public static Result<Integer> assertInRange(int value, int min,
    int max) {
    return assertCondition(value, x -> x >= min && x < max,
        String.format("Assertion error: value %s should be between %s and %s (exclusive)", value, min, max));
}

public static Result<Integer> assertPositiveOrZero(int value) {
    return assertPositiveOrZero(value,
        String.format("Assertion error: value %s must not be negative", 0));
}

public static Result<Integer> assertPositiveOrZero(int value,
    String message) {
    return assertCondition(value, x -> x >= 0, message);
}

public static <A> void assertType(A element, Class<?> clazz) {
    assertType(element, clazz,
        String.format("Wrong type: %s, expected: %s",
            element.getClass().getName(), clazz.getName()));
}
```



```
public static <A> Result<A> assertType(A element, Class<?> clazz,
                                     String message)
{
    return assertCondition(element, e -> e.getClass().equals(clazz)
                           ,message);
}
```

15.2 从文件中读取属性

大多数应用程序都会配置在启动时读取属性文件。属性是键值对，键和值都是字符串。无论选用哪种属性格式（key=value、XML、JSON、YAML 等），程序员总得读取字符串并将其转换为 Java 对象或原始类型。这个过程又无聊又容易出错。你可以使用一个专门的类库来做这件事，但是如果哪里出了错，你就会发现自己抛出异常。为了得到更函数式的行为，你要编写一个自己的库。

15.2.1 载入属性文件

不管你使用了什么格式，过程都一样：读取文件并处理过程中可能产生的任意 IOException。在以下示例中，你会读取一个 Java 属性文件。

要做的第一件事就是读取文件并返回一个 Result<Properties>，如清单 15.2 所示。

清单 15.2 读取一个 Java 属性文件

```
import com.fpjava.common.Result;
import java.io.InputStream;
import java.util.Properties;

public class PropertyReader {

    private final Result<Properties> properties;

    public PropertyReader(String configFileName) {
        this.properties = readProperties(configFileName);
    }

    private Result<Properties> readProperties(String configFileName) {
        try (InputStream inputStream = getClass().getClassLoader()
            .getResourceAsStream(configFileName)) {
            Properties properties = new Properties();
        }
    }
}
```

Result<Properties> 存储在
PropertyReader 类中

通过表示属性文件的字符串创建
PropertyReader 类

文件从类路径中加载

```

    }
    } catch (Exception e) {
        return Result.failure(e);
    }
}

```

通过捕获 Exception 而非 IOException 来处理 InputStream 为 null 的情况^①

① 加载属性文件有可能导致 IOException。注意如果找不到这个文件，并不会导致 IOException，可是生成的 inputStream 会是 null，从而导致 NullPointerException

异常情况下，返回一个包含异常的 Result.Failure

在这个例子里，你从类路径中加载了属性文件。当然它可能从任何地方加载，磁盘、远程 URL 或是其他来源。

15.2.2 将属性读取为字符串

将属性读取为字符串就是简单的用例之一。这样做非常直观。你只需往 PropertyReader 类中增加一个 readProperty 方法，以一个属性名为参数，并返回一个 Result<String>。但请注意以下代码并不能正常工作：

```

public Result<String> getProperty(String name) {
    return properties.map(props -> props.getProperty(name));
}

```

如果属性不存在，getProperty 方法将返回 null。（在 Java 8 中，它应该返回一个 Optional，但是并没有。）请注意，Properties 类可以用默认的属性列表来构造，并且也可以在调用 getProperty 方法自身时传入一个默认值。但是并非所有属性都有默认值。

为了解决这个问题，你可以创建一个辅助方法：

```

public Result<String> getProperty(String name) {
    return properties.flatMap(props -> getProperty(props, name));
}

```

```

private Result<String> getProperty(Properties properties, String name) {
    return Result.of(properties.getProperty(name));
}

```

现在，假设你在类路径中有一个属性文件，包含如下属性：

```
host=acme.org
port=6666
name=
temp=71.3
price=$45
list=34,56,67,89
person=3,Jeanne,Doe
```

你能以安全的方式访问：

```
PropertyReader propertyReader = new PropertyReader("com/fpinjava/
    properties/config.properties");

propertyReader.getProperty("host")
    .forEachOrFail(System.out::println)
    .forEach(System.out::println);

propertyReader.getProperty("name")
    .forEachOrFail(System.out::println)
    .forEach(System.out::println);

propertyReader.getProperty("year")
    .forEachOrFail(System.out::println)
    .forEach(System.out::println);
```

给定属性文件，你就会得到如下结果：

```
acme.org
Null value
```

第一行对应于 `host` 属性，正确。第二行对应于 `name` 属性，它是一个空字符串，可能对也可能不对，你不知道。它取决于业务角度上的名称是否为可选的。第三行对应于缺失的 `year` 属性，但是“`Null value`”并不能提供有用的信息。当然，由于它在赋值于 `year` 变量的 `Result<String>` 中，所以你知道哪个属性缺失。但若消息中有属性名就更理想了。此外，如果文件找不到，你将得到一个什么也说明不了的错误消息：

```
java.lang.NullPointerException
```

15.2.3 生成更好的错误消息

你所面临的问题是一个不应如此的绝佳反面教材。使用 `Java` 标准库，你对一切正常的期望很有自信。尤其是你相信如果找不到或读不了文件时，会得到一个

`IOException`。你甚至希望能被告知文件的全路径，因为“缺失的”文件一般只是文件不在正确的位置（或者文件在正确的位置，只是不在 Java 的搜寻范围中）。在这种情况下，一个好的错误消息应该是：“我在 ‘xyz’ 中搜寻 ‘abc’，但是找不到。”

现在，看看 `ClassLoader.getResourceAsStream` 方法的代码：

```
public InputStream getResourceAsStream(String name) {
    URL url = getResource(name);
    try {
        return url != null ? url.openStream() : null;
    } catch (IOException e) {
        return null;
    }
}
```

不，你不是在做梦。这就是 Java 8 的代码。结论就是：作为程序员的你，不应该不去看 Java 标准库的方法所对应的代码而直接使用它。

请注意，Javadoc 中提到了这个方法返回“读取资源所用的输入流，或者在找不到资源时为 `null`。”这意味着许多事情都可能出错。如果文件找不到可能会发生 `IOException`，或者读取时出问题了；或者文件名为 `null`；或者 `getResource` 方法抛出了一个异常，或者返回 `null`。（看该方法的代码就能明白我的意思。）

你至少应该为每一种情况提供一条不同的消息。尽管事实上不太可能会抛出 `IOException`，你也需要处理这种情况，就好像发生了预期外异常那样的一般情况：

```
private Result<Properties> readProperties(String configFileName) {
    try (InputStream inputStream =
        getClass().getClassLoader().getResourceAsStream(configFileName)) {
        Properties properties = new Properties();
        properties.load(inputStream);
        return Result.of(properties);
    } catch (NullPointerException e) {
        return Result.failure(String.format("File %s not found in classpath",
            configFileName));
    } catch (IOException e) {
        return Result.failure(String.format("IOException reading classpath
            resource %s", configFileName));
    } catch (Exception e) {
        return Result.failure(String.format("Exception reading classpath
            resource %s", configFileName), e);
    }
}
```

现在，如果找不到文件，消息就是

```
File com/fpinjava/properties/config.properties not found in classpath
```

你也需要处理与属性有关的消息。当如下使用代码时

```
Result<String> year = propertyReader.getProperty("year");
```

如果你得到的错误消息是 Null value, 那就很清晰了, 说明没有找到 year 属性。但是在以下示例中, Null value 消息提供的信息并不足以表明缺少了哪个属性:

```
PropertyReader propertyReader =
    new PropertyReader("com/fpinjava/properties/config.properties");
Result<Person> person =
    propertyReader.getProperty("id").map(Integer::parseInt)
        .flatMap(id -> propertyReader.getProperty("firstName")
            .flatMap(firstName -> propertyReader.getProperty("lastName")
                .map(lastName -> Person.apply(id, firstName, lastName)))));
person.forEachOrFail(System.out::println).forEach(System.out::println);
```

为了解决这个问题, 你有几种选择。最简单的就是在 PropertyReader 类的 getProperty 辅助方法中映射该 failure:

```
private Result<String> getProperty(Properties properties, String name) {
    return Result.of(properties.getProperty(name))
        .mapFailure(String.format("Property \"%s\" no found", name));
}
```

上一个示例生成以下错误消息, 清晰地表明了属性文件里找不到 id 属性:

```
Property "id" not found
```

另一个可能导致失败的原因是, 把 id 属性从字符串转换为整型时发生的解析错误。例如, 如果属性是

```
id=three
```

错误消息将会是

```
For input string: "three"
```

这样并没有给你提供有意义的信息, 那是因为它是 Java 8 在解析出错时的标准错误消息。许多标准 Java 错误消息都像这样。例如 NullPointerException。它说有一个引用为 null, 但是并没有说是哪一个。它在这里甚至都没提及遇上了什么错误。异常中带着错误的根源。打印栈轨迹, 你就会得到如下信息:

```
Exception in thread "main" java.lang.NumberFormatException: For input string:
    "three"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.
    java: 48) ...
```

你真正需要的是导致异常的属性名。就像这样：

```
propertyReader.getProperty("id")
    .map(Integer::parseInt)
    .mapFailure(String.format("Invalid format for property \"%s\": ", ???))
```

但是你不能不写两次属性名，并且你还要把“???”替换为找到的值（不可能了，因为值已经丢了）。由于你需要为所有的非字符串属性解析属性值，所以应该将其抽象到 `PropertyReader` 类中。

为了做到这一点，首先你要重命名 `getProperty` 方法：

```
public Result<String> getAsString(String name) {
    return properties.flatMap(props -> getProperty(props, name));
}
```

然后，你要添加一个 `getAsInteger` 方法：

```
public Result<Integer> getAsInteger(String name) {
    Result<String> rString =
        properties.flatMap(props -> getProperty(props, name));
    return rString.flatMap(x -> {
        try {
            return Result.success(Integer.parseInt(x));
        } catch (NumberFormatException e) {
            return Result.failure(String.format("Invalid value while parsing
                                                    property %s: %s", name, x));
        }
    });
}
```

现在，你不必担心转换为整型时的错误了：

```
Result<Person> person =
    propertyReader.getAsInteger("id")
        .flatMap(id -> propertyReader.getAsString("firstName")
            .flatMap(firstName -> propertyReader.getAsString("lastName")
                .map(lastName -> Person.apply(id, firstName, lastName)))));
person.forEachOrFail(System.out::println).forEach(System.out::println);
```

15.2.4 像列表那样读取属性

你可以为其他数字类型照搬整型的处理办法，例如 `long` 或 `double`。但是还可

以更进一步。你可以像列表那样读取属性：

```
list=34,56,67,89
```

只需增加一个专门的方法来处理这种情况。你可以使用以下方法来以整型列表获取一个属性：

```
public Result<List<Integer>> getAsIntegerList(String name) {
    Result<String> rString =
        properties.flatMap(props ->getProperty(props, name));
    return rString.flatMap(s -> {
        try {
            return Result.success(List.fromSeparatedString(s, ',', ')')
                                   .map(Integer::parseInt));
        } catch (NumberFormatException e) {
            return Result.failure(String.format("Invalid value while parsing
                                                property %s: %s", name, s));
        }
    });
}
```

当然，你需要往 List 类中添加 fromSeparatedString 方法。如我在上一章所言，这段代码并不打算使用前几章练习的结果，但是会用本书附带代码 (<https://github.com/fpinjava/fpinjava>) 中提供的 fpinjava-common 模块。它与练习的答案基本相同，但多了一些方法，例如以下示例中的 List.fromCollection(...).

```
public static List<String> fromSeparatedString(String string,
                                                char separator) {
    return List.fromCollection(Arrays.asList(string.split("\\s*"
                                                    + separator + "\\s*")));
}
```

不过还可以更进一步。你可以通过提供转换函数来按任意数值类型列表读取一个属性：

```
public <T> Result<List<T>> getAsList(String name, Function<String, T> f) {
    Result<String> rString
        = properties.flatMap(props ->getProperty(props, name));
    return rString.flatMap(s -> {
        try {
            return Result.success(List.fromSeparatedString(s, ',', ')').map(f));
        } catch (NumberFormatException e) {
            return Result.failure(String.format("Invalid value while parsing
                                                property %s: %s", name, s));
        }
    });
}
```

现在你可以用 `getAsList` 来为各种数字格式定义函数：

```
public Result<List<Integer>> getAsIntegerList(String name) {
    return getAsList(name, Integer::parseInt);
}

public Result<List<Double>> getAsDoubleList(String name) {
    return getAsList(name, Double::parseDouble);
}

public Result<List<Boolean>> getAsBooleanList(String name) {
    return getAsList(name, Boolean::parseBoolean);
}
```

15.2.5 读取枚举值

按 `enum` 值读取属性是一个常见的用例，它是按任意类型读取属性的一个特例。你可以先创建一个把属性转换为任意类型 `T` 的方法，以一个从 `String` 到 `Result<T>` 的函数为参数：

```
public <T> Result<T> getAsType(final Function<String, Result<T>> function,
                               final String name) {
    Result<String> rString =
        properties.flatMap(props -> getProperty(props, name));
    return rString.flatMap(s -> {
        try {
            return function.apply(s);
        } catch (Exception e) {
            return Result.failure(String.format("Invalid value while parsing
                                                property %s: %s", name, s));
        }
    });
}
```

你现在可以通过使用 `getAsType` 来创建一个 `getAsEnum` 方法：

```
public <T extends Enum<?>> Result<T> getAsEnum(final String parameterName,
                                                final Class<T> enumClass) {
    Function<String, Result<T>> f = t -> {
        try {
            T constant = enumClass.getEnumConstants()[0]; ◀ 这是一个小技巧.....
            @SuppressWarnings("unchecked")
            T value = (T) Enum.valueOf(constant.getClass(), t); ◀
            return Result.success(value);
        } catch (Exception e) {
            ◀ 允许在这里使用 T 类
        }
    };
}
```



```

        return Result.failure(String.format("Error parsing property %s: value %s can't be parsed to %s.", t, parameterName, enumClass.getName()));
    }
};
return getAsType(f, parameterName);
}

```

给定以下属性

```
type=SERIAL
```

还有以下 enum,

```

public enum Type {
    SERIAL,
    PARALLEL
}

```

你现在可以用以下代码来读取属性了：

```
Result<Type> type = propertyReader.getAsEnum("type", Type.class);
```

15.2.6 读取任意类型的属性

迄今为止，你已经读取了如下类型的属性：字符串、原始类型（int、double、boolean 等），还有 enum。读取任意类型的属性可能也会很有意思。为此你需要在属性文件中把对象属性写为某种序列化的形式，然后载入这些属性并反序列化之。

你可以用 `getAsType` 方法来读取任意类型的属性。例如，可以读取以下属性以获取一个 `Person`：

```
person=id:3,firstName:Jane,lastName:Doe
```

你要做的全部就是提供一个从 `String` 到 `Result<Person>` 的函数。这个函数应该从字符串 `id:3,firstName:Jane,lastName:Doe` 中创建一个 `Person`。

为了简化使用，你可以创建一个 `getAsPerson` 方法。但是由于它是专门的类型，不应该将其置于 `PropertyReader` 中。可以在 `Person` 类中增加一个以 `PropertyReader` 和属性名为参数的静态工厂方法。

有几种方式都可以实现这个目的。一种方式是按列表获取属性，然后将其拆分为各个元素，并置于 `map` 中的键值对中。这样很容易便能从这个 `map` 中创建一个

Person。另一种方式是创建第二个 `PropertyReader`，它将字符串中的逗号拆分为换行并读取。清单 15.3 展示了 `Person` 类，它有两个从属性字符串中构建实例的特定方法。

清单 15.3 允许按对象或对象列表读取属性的方法

```
public class Person {
    ...
    public static Result<Person> getAsPerson(String propertyName,
                                           PropertyReader propertyReader) {
        Result<String> rString =
            propertyReader.getAsPropertyString(propertyName);
        Result<PropertyReader> rPropReader =
            rString.map(PropertyReader::stringPropertyReader);
        return rPropReader.flatMap(Person::readPerson);
    }

    public static Result<List<Person>> getAsPersonList(String propertyName,
                                                       PropertyReader propertyReader) {
        Result<List<String>> rList =
            propertyReader.getAsStringList(propertyName);
        return rList.flatMap(list -> List.sequence(list.map(s ->
            readPerson(PropertyReader.stringPropertyReader(PropertyReader
                .toPropertyString(s))))));
    }

    private static Result<Person> readPerson(PropertyReader propReader) {
        return propReader.getAsInteger("id")
            .flatMap(id -> propReader.getAsString("firstName")
                .flatMap(firstName -> propReader.getAsString("lastName")
                    .map(lastName -> Person.apply(id, firstName, lastName))));
    }
}
```

`getAsPersonList` 方法允许你读取如下编写的列表属性：

```
employees:\
id:3;firstName:Jane;lastName:Doe,\
id:5;firstName:Paul;lastName:Smith,\
id:8;firstName:Mary;lastName:Winston
```

这些方法需要对 `PropertyReader` 类进行一些改动，参见清单 15.4。

清单 15.4 往 `PropertyReader` 类中增加静态工厂方法

现在由一个 `Result<Properties>` 构造 `PropertyReader`

```
public class PropertyReader {
    private final Result<Properties> properties;
    private final String source;

    private PropertyReader(Result<Properties> properties, String source) {
        this.properties = properties;
        this.source = source;
    }

    ...

    public static String toPropertyString(String s) {
        return s.replace(";", "\n");
    }

    public Result<String> getAsPropertyString(String propertyName) {
        return getAsString(propertyName).map(PropertyReader::toPropertyString);
    }

    private static Result<Properties> readPropertiesFromFile(String
        configFileName) {
        try (InputStream inputStream = PropertyReader.class.getClassLoader()
            .getResourceAsStream(configFileName)) {
            Properties properties = new Properties();
            properties.load(inputStream);
            return Result.of(properties);
        } catch (NullPointerException e) {
            return Result.failure(String.format("File %s not found in classpath",
                configFileName));
        } catch (IOException e) {
            return Result.failure(String.format("IOException reading classpath
                resource %s", configFileName));
        } catch (Exception e) {
            return Result.failure(String.format("Exception reading classpath
                resource %s", configFileName), e);
        }
    }

    private static Result<Properties> readPropertiesFromString(String
        propString) {
        try (Reader reader = new StringReader(propString)) {
            Properties properties = new Properties();
            properties.load(reader);
            return Result.of(properties);
        } catch (Exception e) {
            return Result.failure(String.format("Exception reading classpath
                resource %s", configFileName), e);
        }
    }
}
```

注册了 source 以用作错误消息

这个方法将一个属性值转换为一个属性字符串，以便可以用于嵌套的 `PropertyReader` 的输入

这是原来读取属性文件的方法

该方法读取一个属性并将其转换为一个属性字符串

这是一个从属性字符串中读取属性的新方法

```

    return Result.failure(String.format("Exception reading property
                                       string %s", propString), e);
}
}

public static PropertyReader filePropertyReader(String fileName) {
    return new PropertyReader(readPropertiesFromFile(fileName),
                              String.format("File: %s", fileName));
}

public static PropertyReader stringPropertyReader(String propString) {
    return new PropertyReader(readPropertiesFromString(propString),
                              String.format("String: %s", propString));
}
}

```

该静态属性方法从一个属性字符串创建一个 PropertyReader

该静态工厂方法从一个文件名创建一个 PropertyReader

当然，同样的方法也适用于 XML 属性文件（Java 对此有原生的支持），或者其他格式，如 JSON 或 YAML。

15.3 转换命令式程序：XML读取器

为完成任务编写新的函数式程序固然令人兴奋，但你通常没有时间可以投入。一般来说，你会想要在自己的代码中使用现成的命令式程序。每当你想用 Java 库时都是如此。当然，你可能会发现从头开始构建一个全新的函数式方案更有意思。但你不得不向现实妥协。一般没有实现的时间或预算，你不得不使用现成的非函数式类库。

正如你很快就会发现的，一旦你开始惬意地沉浸在函数式技术中，回到旧的命令式编码风格真的很痛苦。解决方案一般是围绕这些命令式类库构建一个薄的函数式包装。例如，我们将会审视一个用于读取 XML 文件的常见类库，JDOM 2.0.6。它是此类任务最常用的 Java 库。

你将从清单 15.5 中的示例程序开始。该程序来自众多提供关于如何使用 JDOM 的辅导材料的网站之一(<http://mng.bz/4p3x>)的教程。我选择这个例子，是因为它很小，并且很容易印刷在书上。

清单 15.5 用 JDOM 读取 XML 数据：命令式版本

```

import org.jdom2.Document;
import org.jdom2.Element;
import org.jdom2.JDOMException;
import org.jdom2.input.SAXBuilder;
import java.io.File;
import java.io.IOException;
import java.util.List;

public class ReadXmlFile {

    public static void main(String[] args) {
        SAXBuilder builder = new SAXBuilder();
        File xmlFile = new File("path_to_file");
        try {
            Document document = (Document) builder.build(xmlFile);
            Element rootNode = document.getRootElement();
            List list = rootNode.getChildren("staff");
            for (int i = 0; i < list.size(); i++) {
                Element node = (Element) list.get(i);
                System.out.println("First Name : " +
                                   node.getChildText("firstname"));
                System.out.println("\tLast Name : " +
                                   node.getChildText("lastname"));
                System.out.println("\tNick Name : " +
                                   node.getChildText("email"));
                System.out.println("\tSalary : " + node.getChildText("salary"));
            }
        } catch (IOException io) {
            System.out.println(io.getMessage());
        } catch (JDOMException jdomex) {
            System.out.println(jdomex.getMessage());
        }
    }
}

```

这个例子所用的数据文件如清单 15.6 所示。

清单 15.6 待读取的 XML 文件

```

<?xml version="1.0"?>
<company>
  <staff>
    <firstname>Paul</firstname>
    <lastname>Smith</lastname>
    <email>paul.smith@acme.com</email>
    <salary>100000</salary>
  </staff>
  <staff>
    <firstname>Mary</firstname>

```

```
<lastname>Colson</lastname>
<email>mary.colson@acme.com</email>
<salary>200000</salary>
</staff>
</company>
```

首先，你要看看以函数式的方式重写这个例子能为你带来什么收益。你可能遇到的第一个问题是程序没有可重用的部分。当然，这只是一个例子，但即使是例子，也应该以可重用的方式来编写，至少可以测试。在此查看控制台就是唯一测试程序的办法，它会显示期待的结果或是一个错误消息。你将会看到，它甚至可能会显示一个错误的结果。

15.3.1 列出必需的函数

为了让这段程序更加函数式，你应该开始列出所需的基本函数，把它们编写为自治、可重用、可测试的单元，然后通过复合这些函数来编写这个示例。这是程序的主要函数：

1. 读取一个文件并返回 XML 字符串作为内容。
2. 把 XML 字符串转换为元素列表。
3. 把元素列表转换为表示这些元素的字符串列表。

你还将需要一个作用以在电脑屏幕上显示字符串列表。

注意 这段程序的主要函数的描述只适用于可以被整体加载进内存的小文件。

你需要的第一个函数可以实现为如下方法：

```
public static Result<String> readFile2String(String path)
```

该方法不会抛出任何异常，只是返回一个 `Result<String>`。

第二个方法把一个 XML 字符串转换为一个元素列表，所以它需要知道 XML 根元素的名称。其签名如下：

```
private static Result<List<Element>> readDocument(String rootElementName,
                                                    String stringDoc)
```

你需要的第三个方法以一个元素列表为参数并返回一个表示这些元素的字符串

列表。这将由如下签名的方法实现：

```
private static List<String> toStringList(List<Element> list, String format)
```

你最终会需要对数据应用一个作用，所以还需要定义一个如下签名的方法：

```
private static <T> void processList(List<T> list)
```

分解的函数看起来与在命令式编程中所做的区别不大。毕竟，把命令式程序分解为各个单一职责的方法也是一个好的实践。可是，它比看上去的区别更大。请注意，`readDocument` 方法接收的第一个字符串参数是由一个可能（在命令式的世界中）抛出异常的方法返回的。因此，你还需要处理一个附加方法：

```
private static Result<String> getRootElementName()
```

同类的方法也同样能返回文件路径：

```
private static Result<String> getXmlFilePath()
```

值得注意的是，这些函数的参数类型和返回类型不一致！这些函数的命令式版本是偏函数，意味着它们可能会抛出异常，因此其实这是一个显式的转换。抛出异常的方法不能很好地复合。相比之下，你的函数可以完美复合。

15.3.2 复合函数并应用作用

虽然参数和返回值不匹配，但是你的函数可以使用解析式模式轻松复合：

```
final static String format = "First Name : %s\n" +
    "\tLast Name : %s\n" +
    "\tEmail : %s\n" +
    "\tSalary : %s";
...
final Result<String> path = getXmlFilePath();
final Result<String> rDoc = path.flatMap(ReadXmlFile::readFile2String);
final Result<String> rRoot = getRootElementName();
final Result<List<String>> result = rDoc.flatMap(doc -> rRoot
    .flatMap(rootElementName -> readDocument(rootElementName, doc))
    .map(list -> toStringList(list, format)));
```

为了显示结果，你只需应用对应的作用即可：

```
result.forEachOrException(ReadXmlFile::processList)
    .forEach(Throwables::printStackTrace);
```

你的函数式版程序更加整洁，并且完全可测试——或者说当你实现了所有必需的函数后才是。

15.3.3 实现函数

你的程序相当优雅，但是为了让它能够工作，你仍然需要实现所用的函数及作用。好消息是每一个函数都非常简单并容易测试。

首先，你要实现 `getXmlFilePath` 和 `getRootElementName` 函数。在我们的例子中，这些只是常量，而在真正的程序中，需要替换掉它们。

```
private static Result<String> getXmlFilePath() {  
    return Result.of("<path_to_file>");  
}  
  
private static Result<String> getRootElementName() {  
    return Result.of("staff");  
}
```

然后你需要实现 `readFile2String` 方法。可行的实现有很多，这是其中之一：

```
public static Result<String> readFile2String(String path) {  
    try {  
        return Result.success(new String(Files.readAllBytes(Paths.get(path))));  
    } catch (IOException e) {  
        return Result.failure(String.format("IO error while reading file %s",  
                                             path), e);  
    } catch (Exception e) {  
        return Result.failure(String.format("Unexpected error while reading  
                                             file %s", path), e);  
    }  
}
```

请注意你分别捕获了 `IOException` 和 `Exception`。并不是非得这样做，但是它可以让你提供更恰当的错误消息。任何情况下你都需要捕获 `Exception`。（例如，你可能会在此得到一个 `SecurityException`。）

接下来，你需要实现 `readDocument` 方法。该方法以包含 XML 数据的字符串和根元素的名称为参数：

```
private static Result<List<Element>> readDocument(String rootElementName,  
                                                  String stringDoc) {  
    final SAXBuilder builder = new SAXBuilder();  
    try {  
        final Document document =
```



```

    builder.build(new StringReader(stringDoc));
    final Element rootElement = document.getRootElement();
    return Result.success(List.fromCollection(
        rootElement.getChildren(rootElementName)));
} catch (IOException | JDOMException io) {
    return Result.failure(String.format("Invalid root element name '%s'
        or XML data %s", rootElementName, stringDoc), io);
} catch (Exception e) {
    return Result.failure(String.format("Unexpected error while reading XML
        data %s with root element %s", stringDoc, rootElementName), e);
}
}

```

该行可能会抛出一个 `NullPointerException`

该行可能会抛出一个
`IllegalStateException`

你首先要捕获 `IOException`（抛出的可能性极低，因为你只是从一个字符串中读取）和 `JDOMException`，它们都是受检查异常（checked exception），返回一个带有对应错误消息的 `failure`。但是通过查看 `JDOM` 的代码（不先查看一个库方法是如何实现的，就不应该调用它），你看到它可能会抛出 `IllegalStateException` 或 `NullPointerException`。你只好再次捕获 `Exception`。

`toStringList` 方法只需将列表映射到一个负责转换的函数上即可：

```

private static List<String> toStringList(List<Element> list,
                                         String format) {
    return list.map(e -> processElement(e, format));
}
private static String processElement(Element element, String format) {
    return String.format(format, element.getChildText("firstname"),
        element.getChildText("lastname"),
        element.getChildText("email"),
        element.getChildText("salary"));
}

```

最后，你需要实现将会应用于结果的作用：

```

private static <T> void processList(List<T> list) {
    list.forEach(System.out::println);
}

```

15.3.4 让程序更加函数式

你的程序现在更加模块化并易于测试，它的各部分都是可重用的，但是还可以做得更好。你仍然还在使用 4 个非函数式的部分：文件路径、根元素的名称、把元素转换为字符串的格式还有应用于结果的作用。为了让你的程序完全函数式，你应

该让这些部分成为程序的参数。

`processElement` 方法也用了元素名称形式的特定数据, 对应于用于显示的格式化字符串的参数。你可以把格式化参数替换为一个格式化字符串和一个参数列表的 `Tuple`。通过这种方式, `processElement` 方法会变成这样:

```
private static List<String> toStringList(List<Element> list,
                                         Tuple<String, List<String>> format) {
    return list.map(e -> processElement(e, format));
}

private static String processElement(Element element, Tuple<String,
                                         List<String>> format) {
    String formatString = format._1;
    List<String> parameters = format._2.map(element::getChildText);
    return String.format(formatString, parameters.toList().toArray());
}
```

现在你的程序就是一个纯函数了, 接收 4 个参数并返回一个新的 (非函数式的) 可执行程序作为结果。这个版本的程序表示如清单 15.7 所示。

清单 15.7 完全函数式的 XML 读取器程序

现在接收 `Supplier` 的实例作为路径和根元素名。`format` 包括了参数的名称。该方法还接收一个表示可执行程序的附加参数

```
import com.fpinjava.common.*;
import org.jdom2.Document;
import org.jdom2.Element;
import org.jdom2.JDOMException;
import org.jdom2.input.SAXBuilder;
import java.io.IOException;
import java.io.StringReader;
import java.nio.file.Files;
import java.nio.file.Paths;

public class ReadXmlFile {

    public static Executable readXmlFile(Supplier<Result<String>> sPath,
                                         Supplier<Result<String>> sRootName,
                                         Tuple<String, List<String>> format,
                                         Effect<List<String>> e) {
        final Result<String> path = sPath.get();
        final Result<String> rDoc = path.flatMap(ReadXmlFile::readFile2String);
        final Result<String> rRoot = sRootName.get();
```

对 `Supplier` 求值以获取实际的参数

该方法返回一个可执行程序，将 effect 参数应用于 result。请注意这个方法抛出异常。没有什么更好的办法，由于它是一个作用而无法返回一个值

```

final Result<List<String>> result = rDoc.flatMap(doc -> rRoot
    .flatMap(rootElementName -> readDocument(rootElementName, doc))
    .map(list -> toStringList(list, format)));
return () -> result.forEachOrThrow(e); <
}

public static Result<String> readFile2String(String path) {
    try {
        return Result.success(new String(Files.readAllBytes(Paths.get(path))));
    } catch (IOException e) {
        return Result.failure(String.format("IO error while reading file %s",
            path), e);
    } catch (Exception e) {
        return Result.failure(String.format("Unexpected error while reading
            file %s", path), e);
    }
}

private static Result<List<Element>> readDocument(String rootElementName,
    String stringDoc) {
    final SAXBuilder builder = new SAXBuilder();
    try {
        final Document document = builder.build(new StringReader(stringDoc));
        final Element rootElement = document.getRootElement();
        return Result.success(List.fromCollection(
            rootElement.getChildren(rootElementName)));
    } catch (IOException | JDOMException io) {
        return Result.failure(String.format("Invalid root element name '%s'
            or XML data %s", rootElementName, stringDoc), io);
    } catch (Exception e) {
        return Result.failure(String.format("Unexpected error while reading
            XML data %s", stringDoc), e);
    }
}

private static List<String> toStringList(List<Element> list,
    Tuple<String, List<String>> format) {
    return list.map(e -> processElement(e, format));
}

private static String processElement(Element element,
    Tuple<String, List<String>> format) {
    String formatString = format._1;
    List<String> parameters = format._2.map(element::getChildText);
    return String.format(formatString, parameters.toList().toArray());
}
}

```

processElement 方法不再是特定的了

至此，这段程序可以用清单 15.8 展示的客户端代码测试了。

清单 15.8 测试 XML 读取器的客户端代码

```
public class Test {

    private final static Tuple<String, List<String>> format =
        new Tuple<>("First Name : %s\n" +
            "\tLast Name : %s\n" +
            "\tEmail : %s\n" +
            "\tSalary : %s", List.list("firstname", "lastname", "email", "salary"));

    public static void main(String... args) {
        Executable program = ReadXmlFile.readXmlFile(Test::getXmlFilePath,
            Test::getRootElementName, format, Test::processList);
        program.exec();
    }

    private static Result<String> getXmlFilePath() {
        return Result.of("file.xml"); // <- adjust path
    }

    private static Result<String> getRootElementName() {
        return Result.of("staff");
    }

    private static <T> void processList(List<T> list) {
        list.forEach(System.out::println);
    }
}
```

这段程序并不理想，因为你还未处理元素名不合法可能导致的错误。例如，如果使用了一个错误的元素名，可能会得到如下结果：

```
First Name : null
Last Name : Smith
email : paul.smith@acme.com
Salary : 100000
First Name : null
Last Name : Colson
email : mary.colson@acme.com
Salary : 200000
```

看到所有的 **First Name** 都是 **null**，你可以猜测错误是什么，但是要把“**null**”这个词替换为更加明确地包含了错误元素名的消息就更好了。更重要的问题是，如果忘记了列表中的一个元素名，由于以下代码，你将会在 **String.format** 方法中得到一个异常：

```
List<String> parameters = format._2.map(element::getChildText);
return String.format(formatString, parameters.toList().toArray());
```

在这段代码中，参数数组只有 3 个元素而不是 4 个。但是很难从异常跟踪里定位错误的来源。

其实问题的真正原因是你已经把所有的特定数据移出了 `ReadXmlFile` 类，如根元素名、文件路径以及待应用的作用，但是 `processElement` 方法还仍然限定于客户端的业务用例。`ReadXmlFile` 类只允许你读取根元素的所有直接子元素，收集某些直接子元素的值（那些名称和 `format` 一起传入的元素）。

第三个问题是，`readXmlFile` 方法接收两个同样类型的参数。这是参数错误对调的来源，编译器可发现不了。

15.3.5 修复参数类型问题

第三个问题很容易通过使用第 3 章中表述的值类型技术来修复。你可以用 `Result<FilePath>` 和 `Result<ElementName>` 来取代 `Result<String>` 参数。`FilePath` 和 `ElementName` 只是字符串的值类：

```
public class FilePath {
    public final Result<String> value;

    private FilePath(Result<String> value) {
        this.value = value;
    }

    public static FilePath apply(String value) {
        return new FilePath(Result.of(FilePath::isValidPath, value,
                                     "Invalid file path: " + value));
    }

    private static boolean isValidPath(String path) {
        // 替换为校验代码
        return true;
    }
}
```

`ElementName` 类也差不多。当然，如果你想增加一些校验，那就需要增加校验代码。最简单的方式是检查值是否匹配一个正则表达式。为了使用这些新类，`readXmlFile` 方法可以如下修改：

```
public static Executable readXmlFile(Supplier<FilePath> sPath,
                                     Supplier<ElementName> sRootName,
                                     Tuple<String, List<String>> format,
                                     Effect<List<String>> e) {
    final Result<String> path = sPath.get().value;
    final Result<String> rDoc = path.flatMap(ReadXmlFile::readFile2String);
    final Result<String> rRoot = sRootName.get().value;
```

如你所见，变更微乎其微。请注意，如果你觉得公有属性不合适的话，可以在值类型中使用 `getter`。

客户端类也需要修改：

```
private static FilePath getXmlFilePath() {
    return FilePath.apply("<path_to_file>");
}

private static ElementName getRootElementName() {
    return ElementName.apply("staff");
}
```

通过这些修改，现在就不可能在没有编译器警告下交换参数顺序了。

15.3.6 以处理元素的函数为参数

剩下的两个问题可以一次修改解决：把处理元素的函数作为参数传递给 `readXmlFile` 方法。这样的话，该方法就只有一个任务：从文件中读取第一级元素列表，将它们应用于一个可配置的函数，并返回结果。最主要的区别是方法不再生成一个字符串列表并对字符串应用作用了。

你需要让方法泛型化。这意味着只有以下修改：

使这个方法泛型化

`Tuple<String, list<String>>` 的参数消失了，一个新的函数参数取代了它。这就将会应用于把元素列表转换为 `T` 列表的函数

```
public static <T> Executable readXmlFile(Supplier<FilePath> sPath,
                                         Supplier<ElementName> sRootName,
                                         Function<Element, T> f,
                                         Effect<List<T>> e) {
    final Result<String> path = sPath.get().value;
    final Result<String> rDoc = path.flatMap(ReadXmlFile::readFile2String);
    final Result<String> rRoot = sRootName.get().value;
    final Result<List<T>> result = rDoc.flatMap(doc -> rRoot
        .flatMap(rootElementName -> readDocument(rootElementName, doc))
        .map(list -> list.map(f)));
```

待应用的作用现在通过 `List<T>` 参数化了

`toStringList` 和 `processElement` 方法已经删除。它们被一个接收到函数的应用取代了

```
return () -> result.forEachOrThrow(e);
}
```

现在可以相应地修改客户端程序了。这可以把你从用 Tuple 传递 format 字符串和参数名列表的这个小技巧中解放出来：

```
private final static String format = "First Name : %s\n" +
    "\tLast Name : %s\n" +
    "\tEmail : %s\n" +
    "\tSalary : %s";

private final static List<String> elementNames =
    List.list("firstname", "lastname", "email", "salary");

public static void main(String... args) {
    Executable program =
        ReadXmlFile.readXmlFile(Test::getXmlFilePath,
                                Test::getRootElementName,
                                Test::processElement,
                                Test::processList);

    program.exec();
}

private static String processElement(Element element) {
    return String.format(format, elementNames.map(element::getChildText)
        .toList()
        .toArray());
}
...
```

现在 format 又被设置为一个简单的字符串了

元素名列表也被分别设置

processElement 函数作为参数被传递

processElement 方法现在由客户端实现

请注意，processList 作用没有变化。现在由客户端提供一个转换单个元素的函数及一个应用于此元素的作用。

15.3.7 处理元素名称错误

现在你的问题就只剩在读取元素时发生错误了。传递给 readXmlFile 方法的函数返回一个原始值，意味着它应该是一个全函数，但它并不是。在我们一开始的例子中，它因为错误而生成“null”字符串。现在你正使用一个从 Element 到 T 的函数，可以用 Result<String> 作为 T 的实现，但是这不太现实，因为你会得到一个 List<Result<T>>，而需要将其转换为一个 Result<List<T>>。虽然不费吹灰之力，但是毫无疑问应该抽象它。

答案是用一个从 Element 到 Result<T> 的函数，并且用 List.sequence 方法来把结果转换为一个 Result<List<T>>。新方法如下：

```

public static <T> Executable readXmlFile(Supplier<FilePath> sPath,
                                         Supplier<ElementName> sRootName,
                                         Function<Element, Result<T>> f,
                                         Effect<List<T>> e) {
    final Result<String> path = sPath.get().value;
    final Result<String> rDoc = path.flatMap(ReadXmlFile::readFile2String);
    final Result<String> rRoot = sRootName.get().value;
    final Result<List<T>> result = rDoc.flatMap(doc -> rRoot
        .flatMap(rootElementName -> readDocument(rootElementName, doc))
        .flatMap(list -> List.sequence(list.map(f))));
    ...
}

```

这个方法接收的参数现在是从
Element 到 Result<T> 的函数了

结果被“sequence”生成一个 Result<List<T>>。
当然, 你需要把 map 方法更改为 flatMap

唯一新增的变更是在处理元素的方法中处理可能发生的错误。查看 JDOM 的
getChildText 方法的代码方为上策。这个方法实现如下:

```

/**
 * Returns the textual content of the named child element, or null if
 * there's no such child. This method is a convenience because calling
 * <code>getChild().getText()</code> can throw a NullPointerException.
 *
 * @param cname the name of the child
 * @return text content for the named child, or null if no such child
 */
public String getChildText(final String cname) {
    final Element child = getChild(cname);
    if (child == null) {
        return null;
    }
    return child.getText();
}

```

如你所见(随着继续查看 getChild 方法的代码),这个方法不会抛出任何异常,
但是如果元素不存在,它会返回 null。所以你可以修改 processElement 方法:

```

private static Result<String> processElement(Element element) {
    try {
        return Result.of(String.format(format, elementNames.map(name ->
            getChildText(element, name)).toJavaList().toArray()));
    } catch (Exception e) {
        return Result.failure("Exception while formatting element. " +
            "Probable cause is a missing element name in element list " +
            elementNames);
    }
}

```

现在可以用一个自定义的
方法来返回元素文本

为了提供更明确的错误消息,你捕获了在
格式化结果时可能发生的异常


```
private static String getChildText(Element element, String name) {  
    String string = element.getChildText(name);  
    return string != null  
        ? string  
        : "Element " + name + " not found";  
}
```

如果返回值为 null, 将其替换为一个明确的错误

现在, 大部分潜在的错误都以函数式的方式处理了。然而请注意, 还不是所有的错误。如我先前所言, 传递给 `readXmlFile` 方法的作用所抛出的异常不能以这种方式处理。这些是由方法返回的程序抛出的异常。当方法返回程序时, 它还没有被执行。这些异常需要在执行返回的程序时捕获:

```
public static void main(String... args) {  
    Executable program = ReadXmlFile.readXmlFile(Test::getXmlFilePath,  
                                                    Test::getRootElementName,  
                                                    Test::processElement,  
                                                    Test::processList);  
  
    try {  
        program.exec();  
    } catch (Exception e) {  
        System.out.println(e.getMessage());  
    }  
}
```

你可以在本书附带的代码 (<http://github.com/fpinjava/fpinjava>) 中找到完整的范例。

15.4 总结

- 把值置于 `Result` 上下文中等价于函数式的断言。
- 属性文件能够以安全的方式通过 `Result` 上下文被读取。
- 函数式地读取属性将你从处理转换错误中解放出来。
- 可以用一种抽象的方式将属性读取为任意类型、枚举或是集合。
- 可以对遗留的命令式类库构建函数式包装。

附录A

使用Java 8的函数式特性

在 Java 8 发布时，Oracle 宣布它往函数式编程更进了一步。Oracle 的文档《JDK 8 有什么新特性 (*What's New in JDK 8*)》里列举了以下对函数式友好的特性：

- “在此次发布中引入了一个新的语言特性，lambda 表达式 (lambda expression)。它们使你能够将函数作为方法的参数，或是将代码作为数据。lambda 表达式让你能够更加紧凑地表示单一方法接口（请参考函数式接口）的实例。”这是函数式范式的一个非常重要的方面。
- “方法引用 (method reference) 为已经有了名称的方法提供了易读的 lambda 表达式。”这个句子中“已经有了名称的方法”指的可能是“已经存在的方法”，因为不存在没有名称的方法。
- “类型注解 (type annotation) 提供了在任意用到类型的位置上应用注解的能力，而不仅仅是在声明处。”
- “改进了类型推断 (type inference)。”
- “新的 `java.util.stream` 包里的类提供了一个 Stream API 以支持在元素流上进行函数式风格的操作。Collections API 集成了 Stream API，使顺序或并行的 map-reduce 转换成为可能。”

你可以在 Oracle 的原文 *What's new in JDK 8* (<http://mng.bz/27na>) 中阅读这些声

明（还有许多与函数式编程无关的内容）。

在这个文档中，Oracle 并没有列出一些基本要素，而且还遗漏了一个重要事实：

- Function 包
- Optional 类
- CompletableFuture 类
- 事实是，大多数集合都增加了 `stream()` 方法，使其能够转换为 Stream 实例。

所有这些，包括 Optional、Stream 和 CompletableFuture 都是 monad 结构（请参考附录 B）的事实，使 Oracle 的意图变得非常清晰：让 Java 更容易使用函数式编程。

在本书中，我大量使用了某些函数式友好的特性，例如 lambda 和函数式接口，并且我间接地受益于更好的类型推断和扩展的类型注解。尽管如此，我并没有使用其他函数式要素，例如 Optional 和 Stream。我将在本附录中解释原因。

A.1 Optional类

Optional 类与你在第 6 章中开发的 Option 类相似。它应该解决的是 null 引用问题，不过对于函数式编程而言并非一个巨大的帮助。显然哪里搞错了。Optional 类有一个 `get` 方法，如果“包含的”值存在则将其返回，否则返回 null。当然调用这个方法使原来的目标成为泡影。

如果想要使用 Optional 类，那么你应该记住永远也不要调用 `get`。你可能会反对说 Option 类也有 `getOrThrow` 方法，尽管它永远不会返回 null，只会在没有数据时抛出异常。但是这个方法是 `protected`，而且这个类无法被外界继承。这很不一样。这个方法等价于 List 的 `head` 或 `tail` 方法：它们永远不应该从外界被调用。

除此之外，Optional 类也有与 Option 相同的限制：Optional 可以用于真正的可选数据，但是通常数据的缺失是由于错误导致的。Optional 与 Option 一样，并不允许你持有错误的原因，所以它只适用于真正的可选数据，即其数据缺失的原因显而易见时，例如从 `map` 中返回一个值，或者在字符串中的字符位置。如果 `map` 的 `get(key)` 方法没有返回值，无论是 null 还是一个空的 Optional，显然就是找不到 key。如果 `indexOf(char)` 方法没有返回值或是返回空的 Optional，那就是字符串中没有这个字符。

但即使如此也不正确。`map` 的 `get(key)` 方法可以返回 `null` 是因为那个 `key` 里存的就是 `null` 值，或者它可以没有返回值是因为 `key` 为 `null`（假设 `null` 不是一个合法的 `key`）。`indexOf(char)` 方法也可能会有很多原因没有返回值，例如一个负的参数。在这些情况下返回 `Optional` 并不能指向错误的本质。再者，这个 `Optional` 难以复合其他可能出错的方法的返回值。

基于这些理由，`Optional` 正如我们的 `Option` 版本，没有什么用。这正是为什么我们开发了 `Return` 类型，你可以用它来表示可选数据的缺失及错误。

A.2 流

流是 Java 8 的另一个新要素，它混合了三种不同的概念：

- 惰性集合
- `Monad` 集合
- 自动并行化

这三个概念都是独立的，没有混合它们的明显原因。不幸的是，就像许多意在做多件不同事情的工具体那样，它们所做的每一件事都非最优。

`Monad` 数据结构是函数式编程的精华，Java 的集合并不是 `Monad`。只要在集合上调用新增的 `stream()` 方法，你就能创建这样的结构。如果流具有函数式处理所需的所有方法，那就是可行的方案。但是流被设计成可以自动从顺序处理切换到并行处理。这样的处理相当复杂，这可能就是一些重要的方法没有在流中实现的原因。例如，Java 8 的流没有 `takeWhile` 和 `dropWhile` 方法。

也许这就是享用自动并行化需要付出的可以接受的代价，可竟然连这个特性也并非真正有用。（这个问题在 Java 9 中得以解决。）所有的并行流都使用一个 `fork/join` 池，其线程数量与计算机上可用的物理线程减 1（主线程）一样多。任务被分配到池里的每个工作线程的等待队列中。一旦线程用完了自己的任务队列，它将从其他线程中“偷”工作。主线程本身也参与从工作线程中偷工作。

总体上说，结果并不是最优的，因为与此同时，计算机当然可能还会有大量的其他任务。想想一个 Java 企业级应用程序正在接收客户端的请求。这些请求已经被并行处理了，所以进一步并行化每个请求的收益有限。通常在这样的上下文中，完全谈不上有什么收益。

更糟糕的是，由于所有并行流共享了同一个 fork/join 池，如果一个流阻塞了，它可能会阻塞所有其他的流！可以为每一个流专门使用一个池，但是这样会比较复杂，而且只有当你在使用较小的池（意味着较少的线程）时才应该如此。如果你对这样的技术感兴趣，请参阅我在 DZone 上发布的下列文章：

- 《Java 8 犯了什么错，第 III 部分：流和并行流》(<https://dzone.com/articles/whats-wrong-java-8-part-iii>)。
- 《Java 8 犯了什么错，第 VII 部分：还是流》(<https://dzone.com/articles/whats-wrong-java-8-part-vii>)。

关于 Java 8 的流最糟糕的事情很可能是它们只能使用一次。一旦在其上调用了 一个终止方法，它们就无法再被使用了。所有进一步的操作都会生成一个异常。这有两个结果。

第一个是无法记忆化。你只能创建一个新的流，而不能再次访问它。如果值为惰性求值，那么就不得不再次对其求值。

第二个结果更糟糕：Java 8 中的流不能用于解析式模式。想象你打算编写一个函数来检验毕达哥拉斯关系式 $a^2 + b^2 = c^2$ ，用一个如下实现的 Triple 类：

```
public class Triple {
    public final int a;
    public final int b;
    public final int c;
    Triple(int a, int b, int c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    @Override
    public String toString() {
        return String.format("(%s,%s,%s)", a, b, c);
    }
}
```

在命令式 Java 中，pyths 方法可以如下实现：

```
static List<Triple> pyths(int n) {
    List<Triple> result = new ArrayList<>();
    for (int a = 1; a <= n; a++) {
        for (int b = 1; b <= n; b++) {
            for (int c = 1; c <= n; c++) {
```

```

        if (a * a + b * b == c * c) {
            result.add(new Triple (a, b, c));
        }
    }
}
return result;
}

```

“函数式”的版本使用了流，看起来应该像这样：

```

static Stream<Triple> pyths(int n) {
    Stream<Integer> stream = IntStream.rangeClosed(1, n).boxed();
    return stream.flatMap(a -> stream
        .flatMap(b -> stream
            .flatMap(c -> a * a + b * b == c * c
                ? Stream.of(new Triple (a, b, c))
                : Stream.empty()))));
}

```

不幸的是，在 Java 8 中，这样会生成以下异常：

```
java.lang.IllegalStateException: stream has already been operated upon or closed
```

与此相反，你可以用第 5 章中的 List 类来编写这个例子。

另一个 Java 8 的流的限制是：折叠是一个终止操作，意味着一个折叠（Java 8 的流中称为 reduce）将会导致对流中的所有元素求值。为了理解它们的区别，回顾一下你在第 9 章中开发的 Stream.foldRight 方法。通过这个方法，你可以如下编写一个 identity 函数的实现：

```

public Stream<A> identity() {
    return foldRight(Stream::empty, a -> b -> cons(() -> a, b));
}

```

这个方法是完全惰性的，允许你将其用于实现诸如 map、flatMap 以及其他许多方法。用 Java 8 的流完全无法做到。

这是否意味着永远也不应该用 Java 8 的流呢？当然不是。在性能至上的标准下，尤其是当你需要处理原始类型时，Java 8 的流是一个与命令式程序相得益彰的极佳选择。而并行流，应该避免在生产环境中使用。对于大部分的函数式用途，真正函数式的流是一个更好的选择。

如果你想要（或需要）在函数式上下文中使用 Java 8 的 Stream，请知悉尽管 Stream 类型有一个应该用于折叠的 reduce 方法（实际上，这个方法有三个版本），

它并非折叠流的最佳方式。折叠应该在 Collector 的实现中完成。Collector 是一个定义了 5 个方法的接口：

```
@Override
public Supplier<A> supplier();

@Override
public BiConsumer<A, T> accumulator();

@Override
public BinaryOperator<A> combiner();

@Override
public Function<List<List<T>>, List<List<T>>> finisher();

@Override
public Set<Characteristics> characteristics();
```

supplier 方法返回一个单位元的 Supplier<A>。accumulator 方法返回一个 BiConsumer<A, T>，它是折叠函数的非函数式替代品。对应的折叠函数是 BiFunction<A, T, A>，它合并了一个元素和当前的结果。消费者需要将其存放在（Collector 中的）什么地方，而不是返回结果。换句话说，这就是一个折叠的“基于状态变更版”。finisher 是一个可选的函数，它会应用于最终的结果。最后，characteristics 返回一组 Collector 的特征用于优化其工作。特征有三种——CONCURRENT、IDENTITY_FINISH 和 UNORDERED：

- CONCURRENT 表示 accumulator 函数支持并发，可以用于多个线程。
- IDENTITY_FINISH 表示 finisher 函数就是单位元，因而可以忽略。
- UNORDERED 表示流是无序的，允许为并行化带来更多的自由。

以下是把一个 Stream<String> 折叠为一个 List<List<String>> 的 Collector 示例，模拟给定了行最大长度的单词组。首先定义一个通用的 GroupingCollector：

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.*;
import java.util.stream.Collector;

import static java.util.stream.Collector.Characteristics.IDENTITY_FINISH;

public class GroupingCollector<T> {
```

```

private final BiPredicate<List<T>, T> p;

public GroupingCollector(BiPredicate<List<T>, T> p) {
    this.p = p;
}

public void accumulator(List<List<T>> llt, T t) {
    if (! llt.isEmpty()) {
        List<T> last = llt.get(llt.size() - 1);
        if (p.test(last, t)) {
            llt.get(llt.size() - 1).add(t);
        } else {
            addNewList(llt, t);
        }
    } else {
        addNewList(llt, t);
    }
}

public List<List<T>> combiner(List<List<T>> list1, List<List<T>> list2) {
    List<List<T>> result = new ArrayList<>();
    result.addAll(list1);
    result.addAll(list2);
    return result;
}

public static <T> void addNewList(List<List<T>> llt, T t) {
    List<T> list = new ArrayList<>();
    list.add(t);
    llt.add(list);
}

public Collector<T, List<List<T>>, List<List<T>>> collector() {
    return Collector.of(ArrayList::new, this::accumulator, this::combiner,
IDENTITY_FINISH);
}
}

```

然后，为字符串创建一个特定的组收集器：

```

import java.util.List;
import java.util.function.BiPredicate;
import java.util.stream.Collector;

public class StringGrouperCollector {

    private StringGrouperCollector() {
    }

    public static Collector<String, List<List<String>>, List<List<String>>>
        getInstance(int length) {

```



```

BiPredicate<List<String>, String> p = (ls, s) -> length(ls) + s.length() <= length;
return new GroupingCollector<>(p).collector();
}

public static int length(List<String> list) {
    int length = 0;
    for (String s : list) {
        length += s.length();
    }
    return length;
}
}

```

最后，为测试收集器创建客户端代码：

```

public class Client {

    public static void main(String...args) {

        List<String> words2 = Arrays.asList("Once", "upon", "a", "time",
            "there", "was", "a", "prince", "who", "lived", "in", "a", "magnif icent",
            "castle");
        words2.stream().collect(StringGrouperCollector.getInstance(20))
            .forEach(System.out::println);
    }
}

```

这段程序打印如下：

```

[Once, upon, a, time, there]
[was, a, prince, who, lived, in]
[a, magnificent, castle]

```

它的原理与折叠完全相同，对流元素进行抽象迭代。

附录B

Monad

阅读完本书之后，你可能会惊讶地（可能还会失望）发现我并没有提到 Monad。Monad 是一个很火的话题，你能够在网上找到许多所谓的“Monad 教程”。Monad 的话题似乎令人望而生畏，许多程序员不断地阅读这些教程，并希望他们能够最终理解 Monad 是什么。当然，许多程序员理解 Monad，但是只有极少数人能够用简单的语言来解释 Monad。

有那么多 Monad 教程的原因很可能是因为没有任何权威的教程，所以不断有人尝试自己动手翻译或编写。本附录并不是另一个 Monad 教程，我并不打算写一篇关于 Monad 的文章，基于以下两个原因：

- 如果已经读完本书，那你就不需要一份 Monad 教程。虽然我从未用过 Monad 这个术语，但是你已经知道了 Monad 是什么。你知道它的概念并已在本书中大量使用，你只需要为它命名。
- 关于 Monad 有一句神奇的格言：一旦理解了它，你便失去了解释给别人听的能力。

不过让我们看看其他人是怎么说 Monad 的。在网络上搜索，你能找到许多定义：

- Monad 不过就是自函子范畴上的一个幺半群而已。
- Monad 是一些值的一个计算上下文。

- `Monad` 是一个拥有 `unit` 和 `flatMap` 方法的类。

你可能还会找到一些更奇葩的定义：

- `Monad` 是玉米煎饼。
- `Monad` 是大象。

在第一个定义中，定义是合理的……在某些上下文中。第一个定义应该是范畴论的背景中最缜密的定义了。范畴论是一个数学分支，许多程序员对其漠不关心（他们应该关心，不过那是另一回事了）。

对 `Java` 程序员而言，第三个定义可能最容易理解。方法名不重要，重要的是方法需要遵守的规则。

第二个定义可能是最有助于理解 `Monad` 的。`Monad` 是计算上下文，而函数式编程是用函数编程。安全的函数式编程是用全函数编程。不是全函数的函数叫作偏函数，意味着它们并不是总有一个值（见第 2 章）。当没有值时，它们就不高兴了，并开始搞破坏。它们不是纯函数。

思考以下函数：

$$f(x, y) = x + y$$

它是纯函数吗？无人可以回答，这取决于所用的编程语言。在一些语言中，它可能会抛出算数溢出的异常，所以它不是一个全函数，因为它无法对所有 (x, y) 对都有定义。由于抛出异常是一个副作用，因此它不是一个纯函数。

然而，在 `Java` 中使用整型，那它就是纯函数。这意味着无论你传给了函数什么整数对，它总会返回一个值，并且对于相同的整数对这个值总是相同的。因此你可以信任这个函数。这并不是说结果总是正确的。溢出时的结果可能不是你想要的，不过那是另一个问题了。总是会有一个结果（就是说程序不会挂掉），并且这个结果总是一样的。

那么这个函数呢？

$$g(x, y) = x / y$$

在 `Java` 和整型的背景下，即函数以一对整型为参数并返回一个整型。它可能对某些整数对没有值，不是全函数。如果第二个参数为 0，那函数就没有结果并抛出一个异常。这是因为如果把 `g` 视为一个从 $(\text{integer}, \text{integer})$ 到 `integer` 的函数，那么

它并不是一个全函数。

有两种办法可以把 `g` 变成一个全函数：改变定义域，使其成为 `(integer, 非空 integer)` 的函数，或是改变值域，使其成为 `(integer, integer)` 到 `(integer | exception)` 的函数。

为了实现第一种办法，你需要创建一个新的类型：`NonNullInteger`，这是完美可行的。

为了实现第二种办法，你也需要创建一个新的类型：`IntegerOrException`。

函数式程序员倾向于选择第二种办法。

但是如果把函数 `g` 改为返回 `IntegerOrException`，你就不再能够将它与 `f` 复合了。更准确地说，也许你喜欢的这种写法：`f . g (x)` 或 `f(g(x))` 由于类型不再匹配，无法通过编译。

解决方案是创建一个计算上下文，函数可以在其中安全地运行。如果你喜欢隐喻，可以把上下文当成一个安全的盒子。

所以，你需要的就是：

- 一个安全的盒子。
- 把参数值放进盒子的方式。
- 把修改过的函数放进盒子的方式，使其能够应用于参数值。

仅此而已。结果将会是一个包含了函数结果的盒子。

为了在 Java 中举一个简单的例子，你需要稍微修改一下需求，因为 Java 并没有提供这样的安全盒子。它提供了三种安全盒子的类型，但是无一符合我们的情况，因此你要把需求修改为：出错时，结果不是抛异常，而是什么也不返回。（请注意，Java 有一个这样的类型：`Void`。但是要实例化它有些麻烦。）

对于返回结果或空，你可以用在第 6 章中开发的 `Option` 作为安全盒子类型，或者（更好的）是第 7 章中的 `Result` 类型。在标准的 Java 8 中，可以是 `Optional` 类型。

在函数式语言中，把值放入盒子的方法一般命名为 `unit` 或 `return`，不过你在 `Option` 和 `Result` 中将其命名为 `of`，正如 Java 8 的设计师为 `Optional` 所做的。什么东西都不会改变。

允许你将改变后的函数应用于盒子中的值的方法称为 `flatMap`。让我们以一个

简单的函数为例，它接收一个 String 并返回其第一个字符。一个“正常”的方法如下：

```
public static char firstChar(String a) {
    if (a == null || a.length() == 0) {
        throw new IllegalArgumentException();
    } else {
        return a.charAt(0);
    }
}
```

为了使其返回第一个字符或不返回，你需要进行如下改造：

```
public static Optional<Character> firstChar(String a) {
    return a == null || a.length() == 0
        ? Optional.empty()
        : Optional.of(a.charAt(0));
}
```

为了使用这些工具，你需要将一个 String 置于上下文中：

```
Optional<String> data = Optional.of("Hello!");
Optional<Character> character = data.flatMap(ThisClass::firstChar);
```

unit(of) 和 flatMap 方法是让 Optional 成为 Monad 所需的全部。

不仅如此，Monad 的实现中还加入了其他常见的用例。例如，你可能需要使用一个返回原始值（raw value）的函数，如下所示：

```
public static int toUpper(char c) {
    return c >= 'a' && c <= 'z'
        ? c - 32
        : c;
}
```

你可以这样做：

```
Optional<Integer> upperChar = character.flatMap(x -> Optional.of(toUpper(x)));
```

但是效率不高，因为函数将会把结果包装到 Optional 内，而仅仅只是为了 flatMap 方法来打开包装。所以有一个专门用于这种情况的 map 方法：

```
Optional<Integer> upperChar = character.map(ThisClass::toUpper);
```

请注意，如果你要将 map 用于一个返回 Optional 的函数，例如以下代码，你就会获得一个 Optional<Optional<Character>>：

```
Optional<String> data = Optional.of("Hello!");
Optional<???> character = data.map(ThisClass::firstChar);
```

通过称为 `flatten`（或 `join`）的方法，可以将其转换为一个 `Optional<Character>`，可惜的是，`Optional` 里没有这个方法。如你所见，`unit (of)`、`flatMap`、`map` 和 `flatten` 之间密切相关。`flatten` 方法可以如下实现：

```
public static <T> Optional<T> flatten(Optional<Optional<T>> oot) {  
    return oot.flatMap(Function.identity());  
}
```

`Monad` 内还可以抽象出许多其他用例，但是没有必要让一个类型成为 `Monad`。最常用的是 `fold`。这个方法一般被视为容器类型专用，例如 `List` 或 `Stream`，但它不是。例如对于 `Option` 而言，`fold` 可以在 `None<T>` 里这样实现

```
public <U> U fold(U z, Function<U, Function<T, U>> f) {  
    return z;  
}
```

在 `Some<T>` 里这样：

```
public <U> U fold(U z, Function<U, Function<T, U>> f) {  
    return f.apply(z).apply(value);  
}
```

（从技术角度上看，存在一个左折叠，但是它们的区别与这个示例无关。）

你无法把这个方法添加到 `Java 8` 的 `Optional` 类中，它是 `final` 的。但是你可以编写一个外部的实现：

```
public static <T, U> U fold(U z, Function<U, Function<T, U>> f, Optional<T> ot) {  
    return ot.isPresent() ? f.apply(z).apply(ot.get()) : z;  
}
```

你在此使用了非常糟糕的 `Optional.get()`。（上下文的外部禁止访问这个值，也就是说，这个方法不应该是公有的。）这个问题没有巧妙的解决方案。你知道没有值时不应该调用 `get` 方法，所以可以这样编写：

```
public static <T, U> U fold(U z, Function<U, Function<T, U>> f, Optional<T> ot) {  
    return ot.isPresent() ? f.apply(z).apply(ot.orElse(null)) : z;  
}
```

不过这太难看了。以下代码可能是不那么丑陋的实现：

```
public static <T, U> U fold(U z, Function<U, Function<T, U>> f, Optional<T> ot) {  
    return ot.isPresent()  
        ? f.apply(z).apply(ot.get()) : z;  
}
```

```
? f.apply(z).apply(ot.orElseThrow(() ->
    new IllegalStateException("This exception is (never) thrown by
        dead code!")))
: z;
}
```

无论如何，折叠一个 `Optional` 没有用处，除了让人理解 `orElse` 方法其实是一个 `fold`，并且可以如下定义：

```
public static <T> T orElse(Optional<T> ot, T defaultValue) {
    return fold(defaultValue, ignore -> t -> t, ot);
}
```

没错，它完全没有用处，但它有助于在学习 `Stream` 或 `List`（当然不是 `java.util.List`）时理解这类 `Monad`。

还应该往 `Optional` 中添加许多其他缺失的方法，但由于 `Optional` 是 `final` 的，你无法这么做。这是开发一个全新的 `Option Monad` 的一个绝佳理由。与此同时，由于 `Optional` 无法持有数据缺失的原因，使其几乎毫无用武之地。这就是为什么我们需要另一个 `Monad`。在本书中，我们称之为 `Result`，它大体上与 `Scala` 的 `Try` 类相对应。

附录C

敢问路在何方

你现在有了一些用 Java 编写函数式程序的经验。在日常 Java 编程中，应用多少学到的本领取决于你。对于许多 Java 程序员而言，100% 函数式的目标可能太高了。例如，很可能并非每个读者都会打算在生产代码中使用完全函数式的 I/O。但是，如果要对专业项目采用函数式编程范式，你可以有所选择。

C.1 选一门新的语言

第一个选择是你要用到的语言。一般而言，无法选择不同的（更加函数式友好的）语言。不过有时可以。我们才刚刚触及了主题的一些皮毛，用正确的工具可以让你更上一层楼。选择一门函数式语言似乎很复杂，但其实并非如此。如果你已经选择了在领域中更强大的语言，那么切换到其他语言仅仅是有趣而已。如果你已阅读过本书并希望深入，那就不会对弱类型语言感兴趣。所以可能的选择有三：Haskell、Scala 和 Kotlin（也许还有第四个，Frege）。

C.1.1 Haskell

Haskell 是函数式编程事实上的标准语言。Haskell 是一种强类型、惰性的函数式语言，一个有抱负的函数式程序员可能会梦想到的几乎所有特性它都拥有，以

及许多你一开始会很难理解的复杂功能。关于函数式编程的大多数现代的文章和书籍都以 Haskell 为例。此外，他们用的是指定版本的 Haskell：Glasgow Haskell Compiler (GHC)。

无论是否采用你选择的语言，如果你在团队中工作或是需要使用遗留代码，学习 Haskell 都非常有益。当你用 Java 编写函数式程序时，经常要和语言做斗争。用 Haskell 时，你需要与编写命令式程序做斗争。学习 Haskell 将把你的思维训练为函数式思维，没有其他语言能够做到这一点。即使你继续使用 Java，用 Haskell 做原型函数也能受益无穷。

Haskell（对于 Java 程序员而言）的主要问题在于一切都是全新的。你将无法使用任何常规的 Java 工具（除了代码编辑器以外）或是许多惯用的库。当然 Haskell 的库也有很多，但你必须重新学习这一切，包括如何查找、下载和管理它们，如何构建程序，如何处理文档等。

C.1.2 Scala

另一种方案是切换到 Scala。Scala 不是严格意义上的函数式语言。你可以通过 Scala 使用命令式和函数式的风格来编写程序。切换到 Scala 很容易，因为你可以使用类似 Java 的设计来编写 Scala 程序，正如 Java 第一次出现时像编写 C 程序那样来编写 Java 程序。当然，这不是最佳方式，Java 中的许多问题都是由于 C 的传承导致的。随着越来越多的 Java 程序员转换到 Scala，我们将会看到越来越多用这门语言编写的命令式程序。

因此，在 Scala 中编写函数式程序是一门学问，但几乎没有缺失什么功能（如果你使用一些高级函数式库）。而且最大的优点在于，可以重用大部分你已知的东西。你可以在 Eclipse、NetBeans 或 IntelliJ 中编写 Scala 程序。虽然 Scala 有自己的构建工具 (sbt)，你也可以用 Gradle 构建 Scala 程序，甚至可以使用 Maven 或 Ant 来构建 Scala 程序（然而有谁想要这么干？）。此外，你还可以在 Scala 程序中使用所有现存的 Java 库（当然，也可以在 Java 中使用 Scala 库）。如果你需要处理遗留的 Java 代码和工具，这些特性使得 Scala 成为一个绝佳的首选项。

C.1.3 Kotlin

Kotlin 是一门全新的语言，由 IntelliJ IDE 的发行商 JetBrains 设计，这个 IDE

是 Java 及其他许多语言的最佳 IDE。Kotlin 是 Java 本应该成为的语言。它具有许多函数式友好的特性，如函数类型（允许你编写 $(A) \rightarrow (B) \rightarrow C$ 而不是 `Function<A, Function<B, C>>`）、数据类（自动生成构造函数、访问器，还有 `equals` 和 `hashCode` 方法）和隐式方法调用（允许你用 `f(x)`，而非更累赘的 Java 语法 `f.apply(x)` 来调用函数）。此外，Kotlin 与 Java 完全兼容，可以把 Java 和 Kotlin 混合在同一个项目中。由于世界上没有完美的东西（到目前为止），Kotlin 没有函数式集合（即不可变、持久和数据共享），而是使用具有特殊机制的 Java 标准集合——扩展函数（`extension function`）——允许你将方法“添加”到现有的类中（实际上，它允许像调用实例方法那样调用静态方法，并让 `this` 引用“扩展的”实例。）Kotlin 与 Java 能够很好地集成在一起，可以从向 Java 项目中添加 Kotlin 类来迈出第一步。你需要做的全部就是修改构建系统以增加 Kotlin 编译。而对于开发而言，甚至连这都不是必需的，因为 IntelliJ 允许透明地编译和运行 Java/Kotlin 混合项目。在撰写本文时，Kotlin 的版本为 1.0.5，在不久的将来会有很多变化。1.1 版是 beta 版，在你阅读本文时应该已经可以使用了。如果你对 Java 生态系统中的函数式编程感兴趣，那么你确实应该去了解一下。

C.1.4 Frege

另一个有前途的方案是 Frege 语言（以德国数学家和哲学家 Gottlob Frege 命名，发音有些像“frej-guh”）。Frege 是一门非常年轻的语言，对于生产代码而言可能还不够成熟，但它正在快速发展，并且可能成为 JVM 上纯函数式编程的首选语言。Frege 实际上是“JVM 上的 Haskell”，它与 Haskell 非常类似，同时还保有使用所有现有 Java 库的能力。因为可以与 Java（类似 Scala）混合使用，所以它成为平滑过渡的一个绝佳选择。如果你决定将 Haskell 或 Kotlin 作为原型语言学习，为什么不试试 Frege？关于 Frege 的更多信息，请访问 <https://github.com/Frege/frege> 和 <http://fregepl.blogspot.fr/>。

C.1.5 动态类型函数式语言怎么样

动态类型的函数式语言不同于前面所说的那些，它们并不依靠类型系统来帮助程序员编写正确的程序，而是把程序员从类型的专制中解放出来，允许他们编写类型不正确但又可以编译的程序。

为了让它听上去像是一个优点，这种语言通常被称为“动态类型语言”。每个人都知道，动态比静态更好，因此质量应该是它的特色。不幸的是，与“强类型”语言（如 Java、Haskell 或 Scala）相比，这些语言都被称为“弱类型”。并不是说弱类型的语言不好。它们只是有一个非常重要的区别：如果混淆了类型，编译器一般不会警告你，程序只会在运行时崩溃。这是一个选择，由你自己决定。

C.2 继续Java

你可以继续使用 Java。为了从学习函数式范式到应用于 Java 生产代码中顺利过渡，你需要一个 Java 函数式库。虽然可以用你在阅读本书时开发的程序，但需要注意，维护一个库是一项庞大的任务。如果你是唯一的用户，这可能是一个最佳选择，因为你可以根据自己的需求来定制这个库。每当发现一个可以被抽象到库中的新函数，你就可以放手去做。但如果你在一个团队中工作，那就是另一回事了。你必须顾及每个人的需求，小心不要破坏任何东西，并且始终向后兼容。这可不是一件容易做到的事情。

还有一种选择是使用一个由许多人开发和测试的现有开源库。你不会有同样的自由来添加所需的新功能，但是你可以立刻收到成效。如果真的想要一个新特性，你可以自己添加并提交给社区。

C.2.1 Functional Java

Functional Java 是最早的开源 Java 函数式库之一，至今仍在使用。它早于 Java 8，一开始使用匿名类来表示函数。这可能是最原教旨主义的函数式库。如果你打算成为一个原教旨主义的函数式程序员，这是一个好东西。即使你不是，使用它并查看它的源代码也可以得到非常宝贵的经验。但是请注意，文档很少。虽然你在本书中学到的东西会很有帮助，但你也需要自己搞清楚如何使用它。

还要注意的是，许多杰出的函数式程序员开发了这个库，其中的一些程序员现在已经把兴趣转向更加函数式且友好的语言上了。你可以在这个网站上找到更多信息：<http://www.functionaljava.org/>。

C.2.2 Javaslang

Javaslang 是 Java 较新的一个不太极端的函数式库。它拥有更好的文档，涵盖了

基本示例,虽然文档只是一个(庞大的)单页面。这里再说一遍,在使用 `JavaSlang` 时,你在本书中学到的内容会很有帮助。正如我所说, `JavaSlang` 的做法不那么原教旨主义,这可能使它更容易过渡,尤其是对函数式范式中兴趣不定的团队。有一个小毛病:尽管它有流,但是这些流也遇到了和 Java 8 中的流的同样问题:它们没有惰性折叠。但是,这个问题表明了确实有实现它们的计划。另一方面,它提供了可用的模式匹配机制。你可以在 <http://javaslang.io/> 上找到有关该库的信息。

C.2.3 Cyclops

`Cyclops` 扮演着“JDK 8 的强大、轻量和模块化扩展”的角色,但它不止于此。事实上,它是 Java 的完整的函数式库,还提供额外的支持,以利用标准的 Java 数据类型,使其真正有用。例如,它为标准的 Java 集合增加了函数式方法,并且还提供了不可变的持久化集合,正如你在本书中开发的那样。`Cyclops` 还为 Java 8 的 `Stream` 接口提供了缺失的方法,例如 `takeWhile` 和 `dropWhile`。`Cyclops` 确实充满了有意思的东西,如可重放的流、记忆化、trampolining、模式匹配、元组等。并且它拥有可能在所有能用的 Java 函数式库中最好的文档。最后,它设计为可以与其他库一起工作,如 `Functional Java` 或 `JavaSlang` (甚至 `Guava`)。`Cyclops` 可以在 <https://github.com/aol/cyclops> 上找到。

C.2.4 其他函数式库

曾经还有其他 Java 函数式库,如 `Fun4j`、`LambdaJ`、`op4j` 和 `Apache Commons Functor`。所有这些库都早于 Java 8,自从发布 Java 8 以来,它们都不再发展了,主要是因为 Java 8 令它们过时了。

`Guava` 还在不断发展,因为它不是一个函数式的库,而是一个包含了函数的库。但是,`Guava` 的函数式特性并没有很大进展,现在已经过时了。

C.3 进一步阅读

如果想要了解有关函数式编程的更多信息,你可以在互联网上找到大量资源。许多文章和书籍都是针对函数式编程而写的,但 Java 中的函数式编程并不多。但是,你可能会发现一些有用的文章写的是关于一般的函数式编程,其中包含“函数式语言”中的示例,因为许多概念都适用于 Java。

以下是你可能会感兴趣的文章的不完整清单：

John Hughes, “Why Functional Programming matters,” from “Research Topics in Functional Programming,” ed. D. Turner (Addison-Wesley, 1990), <http://mng.bz/qp3B>.

这篇非常有意思的文章主要与高阶函数和惰性有关，并解释了为什么这些特性对于编写更好、更安全的程序是如此重要。

Philip Walder, “Theorems for free!” (University of Glasgow, 1989), <http://mng.bz/my25>.

这篇文章很难读懂，但如果你想确定一个强大的类型系统可以对身为程序员的你提供些什么，那么值得付出努力。

Chris Okasaki, “Purely Functional Data Structures” (thesis, School of Computer Science, Carnegie Mellon University, 1996), <http://mng.bz/8Gz4>.

这篇容易阅读的大学论文与如何构建纯函数式的数据结构有关。示例是用函数式语言 Standard ML 编写的。Okasaki 编写了一本基于该论文的书，更容易阅读并拥有 Haskell 的示例。如果对函数式的（不变的和持久化的）数据结构感兴趣，那它就是你必须读的一本书籍。

Kimball Germane and Matthew Might, “Deletion: the curse of the red-black tree,” JFP 24, 4 (2014): 423–433, <http://mng.bz/yl57>.

本文补充了 Okasaki 对函数式红黑树的介绍。在他的书中，Okasaki 并没有实现从这个结构中删除元素，而是将其作为练习留给读者。这篇文章就是关于删除的实现。

Graham Hutton, “A tutorial on the universality and expressiveness of fold,” J. Functional Programming 9, 4 (1999): 355–372, <http://mng.bz/me7Z>.

这是关于函数式编程的最有趣的文章之一，非常容易阅读。如果你想完全理解折叠，请勿错过。

Ralf Hinze and Ross Patterson, “Finger trees: a simple general-purpose data structure,” <http://mng.bz/AYZS>.

这是关于非常有趣的函数式数据结构的一篇文章，函数式数据结构允许所有类型的访问和操作都具有良好的性能，尽管并不像标题所说的那么简单。在 Java 中实现它是一个受益匪浅的挑战（有几个著名的实现。）

有一个大胆的论断：学习函数式编程，你会变成一个更好的程序员。幸运的是，无须掌握函数式编程的方方面面，你就能够有所获益。只要消化了几个主要原则，你的代码立即就能在可扩展性、可读性和可维护性等方面更上一层楼。同时 bug 的数量也会减少！让我们赶紧开始吧！

《Java 函数式编程》教你如何让新代码和既存代码都能够高效运行。本书使用了易于掌握的例子、练习和图表来教授函数式编程的核心原则，诸如引用透明性、不可变性、持久化和惰性。与此同时，你还将发现 Java 8 中受函数式编程启发的哪一个新特性将会对你有所助益。

本书内容

- 编写更加易读和易推断的代码
- 更安全地并发和并行编程
- 无须异常即可处理错误
- Java 8 的特性，诸如 lambda、方法引用和函数式接口

本书面向先前没有函数式编程经验的 Java 开发者。

Pierre-Yves Saumont 是一名拥有三十年设计和构建企业级软件经验的 Java 开发者。他目前是 Alcatel-Lucent Submarine Networks 公司的一名软件研发工程师。



博文视点Broadview



@博文视点Broadview



MANNING



策划编辑：张春雨
责任编辑：刘 舫
封面设计：吴海燕

Java函数式编程

一份给Java程序员的卓越的函数式编程详解。

——Piotr Bzdyl, SmartRecruiters

帮助我理解了函数式编程的基本概念。

——Philippe Charrière, GitHub

用Java这样的大众编程语言来帮助解释函数式编程的基本概念和思维模式。

——Sebastian Metzger, snapADDY

向你展示如何编写现代的Java代码。

——Alessandro Campeis, Vimar

上架建议：编程语言/Java

ISBN 978-7-121-33021-6



9 787121 330216 >

定价：119.00元